António Mendes Rosa

Licenciado em Engenharia Eletrotécnica e de Computadores

Análise de Fluxos em Tempo Real para Gestão de Dados de Tráfego

Dissertação para obtenção do Grau de Mestre em

Engenharia Eletrotécnica e de Computadores

Orientador: Ricardo Luís Rosa Jardim Gonçalves, Professor Associado com Agregação, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Co-orientador: Ruben Duarte Dias da Costa, Investigador Sénior no Uninova, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente: Professor Doutor Luís Augusto Bica Gomes de Oliveira
Arguente: Professor Doutor Tiago Oliveira Machado de Figueiredo Cardoso
Vogal: Doutor Ruben Duarte Dias da Costa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2017

Análise de Fluxos em Tempo Real para Gestão de Dados de Tráfego

Copyright © António Mendes Rosa, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Para os meus pais, irmã e namorada.

Agradecimentos

O curto espaço dedicado a estes agradecimentos não é suficiente para agradecer a todas as pessoas, que ao longo do meu Mestrado Integrado em Engenharia Eletrotécnica e de Computadores, contribuíram, direta ou indiretamente, para cumprir os meus objetivos e concluir mais uma etapa da minha vida académica. Ainda assim, deixo apenas algumas palavras, como prova do meu sentido e profundo agradecimento.

Queria agradecer antes de mais ao meu orientador, o Professor Ricardo Gonçalves, que me concedeu a oportunidade de trabalhar num projeto do Group for Research in Interoperability of Systems (GRIS) e acreditou em mim para a conclusão do mesmo.

Ao meu co-orientador, o Professor e investigador sénior no instituto UNINOVA, Ruben Costa, devo um especial obrigado pois sem o seu apoio e as inúmeras reuniões, não teria sido possível a concretização do projeto. Queria também agradecer à sua equipa, particularmente ao Paulo Alves Figueiras e ao Guilherme Guerreiro pela contribuição incansável no projeto.

Aos meus colegas de curso, Daniel Alves, Hugo Antunes e Ana Filipa Sebastião, por terem feito com que este percurso se tornasse mais fácil e divertido, e pela ajuda que me deram nos momentos mais difíceis ao longo do mesmo.

Aos meus pais, por me terem tornado na pessoa que sou hoje e pelo incondicional apoio ao longo da minha vida, e também ao longo deste projeto, enchendo-me de motivação e querer para finalizar esta etapa. À minha irmã um especial obrigado pela paciência e compreensão nos momentos mais complicados deste longo percurso.

Ao meu mais que tudo, Carolina, que sempre me apoiou e me fez acreditar que este trabalho se tornaria realidade.

Resumo

Com a explosão do conceito de *Big Data* e com a evolução da tecnologia no geral, inúmeros setores estão a mudar profundamente de paradigma. Um desses setores é o setor dos transportes. Com esta quantidade massiva de dados a ser produzida, tornou-se fulcral a sua análise profunda e em tempo real para melhorar a forma como nos deslocamos.

Um dos maiores problemas neste setor é o congestionamento rodoviário que afeta milhares de pessoas diariamente, principalmente nas grandes metrópoles. O objetivo deste trabalho é o estudo das metodologias e ferramentas existentes para tratar e processar uma grande quantidade de dados de tráfego em tempo real. Desde metodologias de tratamento e gestão de grandes volumes de dados, técnicas de processamento em tempo real e de visualização, até à implementação de um protótipo que aplica algumas destas técnicas.

Com este processamento e conhecimento adquirido, pretende-se ter uma maior noção do tráfego em tempo real de veículos e de padrões de trânsito, para a partir destes se poderem tomar decisões importantes e que terão um impacto direto na vida dos utilizadores, como o aumento da segurança rodoviária ou a melhoria da qualidade de toda a experiência de viagem.

A aplicação protótipo desenvolvida pode muito bem ser vista como a base de uma aplicação para monitorização de tráfego em tempo real.

Palavras-chave: Grandes Volumes de Dados, Sistemas de Gestão de *streams*, Sistemas de Transporte Inteligentes, Processamento em tempo real, Prospeção de Dados, Visualização de Dados

Abstract

With an explosion of the *Big Data* concept and with the development of the technology in general, countless sectors are facing a paradigm shift. One of the sectors is the intelligent transportation sector. With this massive amount of data being produced, it has become central to its in-depth, real and non-real time analysis, to improve the way we travel.

One of the biggest problems in the sector is the traffic congestion that affects thousands of people every day, especially in big cities. The purpose of this work is to study the existing mechanisms and methodologies to minimize the problem, from methodologies for treatment or manage large volumes of data, techniques of real time processing and visualization of data, to the actual implementation of an application that reunite all of these tecnhiques.

With this processing and knowledge acquired, we want to have a greater sense of real-time traffic congestion and patterns, to take more wisable decisions that can increase the security and the quality of all the travel experience.

The developed application may well be seen as a basis for an application for real-time traffic monitoring.

Keywords: Big Data, Data Stream Management Systems, Intelligent Transportation Systems, Real time processing, Data Mining, Data Visualization

Índice

Lista de Figuras xv

Lista de Tabelas xvii

Glossário xix

Siglas xxii

1 Introdução 1
- 1.1 Motivação . . . 1
- 1.2 Contexto . . . 4
  - 1.2.1 Contextualização do Problema . . . 5
- 1.3 Problema . . . 8
- 1.4 Abordagem . . . 9
  - 1.4.1 Contribuições . . . 11
- 1.5 Estrutura do Documento . . . 12

2 Estado da Arte 13
- 2.1 Técnicas/Tecnologias de *Data Processing* . . . 13
  - 2.1.1 Data Stream Management Systems . . . 14
  - 2.1.2 Sistemas Distribuídos de *Stream Processing* . . . 17
  - 2.1.3 Ferramentas de *Stream Processing* . . . 20
- 2.2 CRISP-DM . . . 29
- 2.3 Trabalho Relacionado . . . 32

3 Fontes de Dados 37
- 3.1 Data Understanding . . . 37
  - 3.1.1 Descrição dos Dados . . . 37
  - 3.1.2 Qualidade e Disponibilidade dos Dados . . . 40
- 3.2 Data Selection . . . 46
  - 3.2.1 Limpeza de Dados e Seleção Final . . . 46
- 3.3 Exploração dos Dados . . . 49

4 Implementação 57

4.1 Tecnologias 57
4.2 Arquitetura Desenvolvida 59
4.2.1 Ingestão e Modelação de Dados 59
4.2.2 Processamento de Dados 62
4.2.3 Visualização de Dados 65
4.3 Workflow da Aplicação 69

5 Validação e Resultados **71**
5.1 Metodologia de Teste 71
5.2 Resultados 73
5.2.1 Resultados dos testes referentes à Ingestão e ao Processamento de *streams* em tempo real 73
5.2.2 Resultados da Visualização de Dados 75
5.3 Discussão de Resultados 77

6 Conclusão **81**
6.1 Trabalho Futuro 82

Bibliografia **85**

Lista de Figuras

1.1 Previsão e crescimento do Volume de Dados (2005 - 2019) 1
1.2 Comparação entre os vários Sistemas de Gestão de Dados 7
1.3 Metodologia CRISP-DM para data mining 9

2.1 Modelo Sistema Aurora 17
2.2 Modelo geral de fluxo de dados de uma plataforma de *stream processing* . . . 18
2.3 Topologia Apache Storm 21
2.4 Arquitetura Cluster Apache Storm 22
2.5 Arquitetura dos *worker nodes* 23
2.6 Arquitetura *Supervisor Node* - Apache Storm 24
2.7 Fluxo das mensagens no *worker Node* - Apache Storm 25
2.8 Modelo DStream *Discretized Streams* - Spark Streaming 26
2.9 Arquitetura geral cluster Spark Streaming 27
2.10 Partition Stream - Samza 27
2.11 Etapas que compõem o KDD 29
2.12 Modelo Processual CRISP-DM 30

3.1 Exemplo Notação JSON 38
3.2 Localização e distribuição geográfica - Sensores Eslovénia 39
3.3 N.º de Registos por mês desde Janeiro 2016 até Maio 2017 41
3.4 N.º registos de um único sensor no mês de Abril de 2017 43
3.5 Comparação entre a diferença de tempo entre veículos e a ocupação da via . 43
3.6 Relação entre o estado do tráfego e a ocupação da via 44
3.7 Contraste entre velocidade média e ocupação da via 45
3.8 Comparação entre a descrição do estado do tráfego e a ocupação da via . . . 46
3.9 Comparação entre os registos antes e após a limpeza dos dados 47
3.10 Formato dos dados no MongoDB após processo de limpeza e seleção 48
3.11 Ocupação média por mês de sensores localizados em estradas com comunicação com outros países - Ano 2016 49
3.12 Localização dos sensores utilizados para analisar o padrão anual de ocupação - Ano 2016 50
3.13 Ocupação média por hora de todos os sensores - Ano 2017 51
3.14 Ocupação média por semana (14 - 22 de Fevereiro) - Ano 2016 51

3.15 Análise do dia de feriado nacional da Eslovénia (8 Fevereiro) e do primeiro dia do ano (1 de Janeiro) de 2017 . . . 52
3.16 BoxPlot 2016 - Ocupação média . . . 54
3.17 BoxPlot (Janeiro - Maio) 2017 - Ocupação média . . . 54
3.18 Localização geográfica dos sensores no acesso a Liubliana . . . 55
3.19 BoxPlot 2016 - Velocidade média . . . 55
3.20 BoxPlot (Janeiro - Maio) 2017 - Velocidade média . . . 56

4.1 *Lambda Architecture* . . . 59
4.2 Arquitetura geral da aplicação . . . 59
4.3 Arquitetura do sistema de *queues* . . . 60
4.4 Processo de inserção de dados na fila de mensagens do RabbitMQ . . . 61
4.5 Arquitetura da topologia do Apache Storm . . . 63
4.6 Arquitetura da topologia Storm para visualização em tempo real . . . 67
4.7 Modelo de troca de dados entre clientes JMS . . . 67
4.8 Arquitetura da ligação Apache Storm - Browser: Apache Camel . . . 68
4.9 Workflow topologia Apache Storm . . . 70

5.1 Topologia utilizada para os testes de validação da topologia Storm . . . 72
5.2 Número médio de mensagens inseridas na fila do RabbitMQ para configuração com 1 *worker* . . . 73
5.3 Tempos de execução médios para todas as unidades de processamento do Apache Storm para configuração com 1 *worker* . . . 73
5.4 Número médio de mensagens inseridas na fila do RabbitMQ para configuração com 4 *workers* . . . 74
5.5 Tempos de execução médios para todas as unidades de processamento do Apache Storm para configuração com 4 *workers* . . . 74
5.6 Tempo de execução para processar todos os dados dos 17 meses . . . 74
5.7 Visualização da ocupação média horária das estradas na Eslovénia a partir da elevação de barras localizadas nos sensores em um mapa, utilizando a API leaflet . . . 75
5.8 Visualização da ocupação e velocidade por direção e por sensor, utilizando a API *Highcharts* . . . 75
5.9 Visualização da ocupação e velocidade de um subconjunto de sensores na Eslovénia a partir da mudança de raio e cor de círculos, utilizando a API *leaflet* 76
5.10 Visualização da comparação entre a ocupação e velocidade por direção de um subconjunto de sensores na Eslovénia, utilizando a API *Highcharts* . . . 76

Lista de Tabelas

2.1 Comparação entre Data Stream Management System (DSMS) e Database Management System (DBMS) 15
2.2 Apache Storm vs Samza vs Spark Streaming 28

3.1 Descrição dos meta-dados 40
3.2 Percentagem por mês da disponibilidade dos dados 42
3.3 Tradução do estado do tráfego rodoviário de Esloveno para Português 45

4.1 Principais tecnologias utilizadas 58

5.1 Parametrização da topologia a ser testada 72

Glossário

analytics — Analytics refere-se à habilidade de utilizar dados, análises e raciocínio sistemático para conduzir a um processo de tomada de decisão mais proveitoso. Existem diversos tipos de análise que compõem o termo, como previsão (*Forecasting*), data mining, text mining, modelagem estatística entre outras.

cluster — Um cluster consiste num sistema que relaciona dois ou mais computadores de forma a que estes trabalhem juntos com o intuito de processar uma tarefa.

clustering — Clustering é uma técnica de Data Mining para fazer agrupamentos automáticos de dados tendo em conta o seu grau de semelhança. O critério de semelhança faz parte da definição do problema e, por isso, o algoritmo depende do mesmo.

daemon — Em sistemas operativos *multitasking*, um daemon é um programa de computador que é executado num plano secundário sem estar acessível ao controlo direto do utilizador.

data mining — É o processo de explorar grandes quantidades de dados através de algoritmos, à procura de padrões consistentes, como regras de associação ou sequências temporais, para detetar relacionamentos sistemáticos entre variáveis, detetando assim novos subconjuntos de dados.

framework — É uma plataforma para desenvolver aplicações de software, fornece uma base sobre a qual os desenvolvedores de software podem criar programas para uma plataforma específica.

JAR — JAR (Java ARchive) é um arquivo/ficheiro compactado usado para distribuir um conjunto de classes Java, um aplicativo Java, ou outros itens como imagens, XMLs, entre outros. É principalmente usado para armazenar classes compiladas.

Map Reduce — É um modelo de programação desenhado para processar grandes volumes de dados em paralelo, dividindo o trabalho num conjunto de tarefas independentes.

meta-dados	São dados acerca de outros dados. Um item de um meta-dado pode dizer do que se trata aquele dado, geralmente uma informação inteligível por um computador. Os meta-dados facilitam o entendimento dos relacionamentos e a utilidade das informações dos dados.
middleware	Middleware ou mediador em português no campo da computação distribuída é um *software* que faz a mediação entre o sistema operativo e bases de dados ou aplicações entre software e outras aplicações.
streaming de dados	Dados em streaming são dados gerados continuamente por milhares de fontes de dados, que geralmente enviam os registos de dados simultaneamente, em tamanhos pequenos (na ordem dos *Kilobytes*).
tuples	É uma lista finita ordenada de elementos.

Siglas

API	Application Programming Interface.
CQL	Continuous Query Language.
CRISP-DM	Cross Industry Standard Process for Data Mining.
DAG	Directed Acyclic Graph.
DAHP	Database-Active Human-Passive.
DBMS	Database Management System.
DSMS	Data Stream Management System.
DSW	Data Stream Warehouses.
GFT	Google Flu Trends.
GPS	Global Positioning System.
GRIS	Group for Research in Interoperability of Systems.
HADP	Human-Active Database-Passive.
HMM	Hidden Markov Model.
IBM	International Business Machines.
ITS	Intelligent Transportation Systems.

JMS	Java Message Service.
JVM	Java Virtual Machine.
KDD	Knowledge Discovery in Database.
MIT	Massachusetts Institute of Technology.
NTMA	Network Traffic Monitoring and Analysis.
RDBMS	Relational Database Management System.
RDD	Resilient Distributed Dataset.
RFID	Radio-Frequency IDentification.
SPL	Stream Processing Language.
SPOF	Single Point of Failure.
SQL	Structured Query Language.
SQuAL	Stream Query Algebra.
TCP	Transmission Control Protocol.
V2I	Vehicle-to-Infrastructure.
V2V	Vehicle-to-Vehicle.

Capítulo 1

Introdução

1.1 Motivação

Num futuro mais breve do que aquele que se imagina, todos os objetos no planeta Terra produzirão dados, incluindo as nossas casas, carros e até o nosso corpo, fazendo com que tudo o que fazemos atualmente deixe um rasto no mundo digital que irá permanecer muito para além da nossa vida. Estima-se que até 2020 o volume de dados atinja os 40 *Zettabytes* [14], para se ter uma ideia da grandeza, se somássemos todos os grãos de areia existentes em todas as praias do planeta e multiplicássemos esse valor por 57, obteríamos os 40 *Zettabytes* de informação [26]. Como podemos observar na Figura 1.1, desde 2011 o volume de dados tem vindo a aumentar exponencialmente e o mais incrível é que menos de 0,5% destes dados são analisados e utilizados, daí a potencialidade por explorar [36].

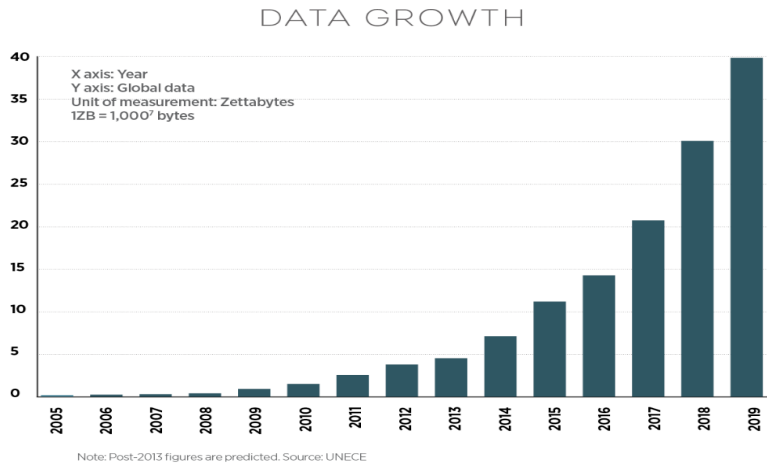


Figura 1.1: Previsão e crescimento do Volume de Dados (2005 - 2019) [1]

A esta grande quantidade de dados, bem como às técnicas e tecnologias que foram desenvolvidas para os analisar, dá-se o nome de *Big Data*. A ideia é mergulharmos nos dados através de técnicas que permitam a deteção de padrões, que à partida não seriam possíveis com apenas a inspeção humana, técnicas essas que tipicamente se denominam por data mining. Há tantas oportunidades para melhorar a qualidade de vida das populações, que simplesmente resolver não analisar os dados de que dispomos atualmente se tornou insustentável. Existem inúmeros setores onde a análise a estes dados é imprescindível, tais como a saúde, segurança, manufatura, transportes, educação, entre muitos outros.

O *Big Data* é um paradigma recente na sociedade, contudo o conceito por detrás do mesmo, que se traduz na ação de reunir e armazenar grandes quantidades de dados, ter surgido há muito tempo atrás. Este paradigma foi motivado pela crescente diversidade e quantidade de dados que tem vindo a ser produzida de variadíssimas fontes. Mas o importante não é a quantidade, mas sim o que decidimos fazer com estes dados. O *Big Data* é baseado no modelo dos 5 V's: volume, variedade, velocidade, veracidade e valor; que serão seguidamente explicitados.

Volume pode ser justificado pelas atividades que realizamos no quotidiano. Diariamente existe um grande volume de troca de *e-mails*, transações bancárias, interações em redes sociais, registos de chamadas, tráfego de dados em linhas telefónicas, entre outras, e que geram uma imensidão de dados. Verdadeiramente o volume é o atributo que melhor descreve o conceito de *Big Data*, e está por isso sempre a ele associado.

Variedade, pois se temos um enorme volume de dados, consequentemente obtemos uma grande variedade dos mesmos. Atualmente são produzidos dados dos mais variados tipos, nomeadamente *csv*, JSON, *access*, vídeo, imagem, *pdf* ou até mesmo ficheiros de texto. O mundo real produz dados diversificados e sem qualquer tipo de estrutura facilmente identificável, o que leva a que um dos grandes desafios do *Big Data* seja saber interpretá-los de forma a atribuir-lhes um significado.

Velocidade, na verdade vivemos num tempo em que as notícias de ontem já se encontram desatualizadas. De facto, o movimento de dados é praticamente efetuado em tempo real e o tempo de atualização dos mesmos é muito curto.

Veracidade, pois para que possam ser tiradas ilações e tomadas decisões vantajosas ao negócio ao analisar os dados, estes têm de estar de acordo com a realidade. Efetivamente o conceito de velocidade, descrito anteriormente, se correlaciona com o de veracidade, pela necessidade constante de análise em tempo real. Isto significa análise de dados que estão a ser produzidos no momento, pois dados passados podem não ser considerados verdadeiros para o momento em que vão ser analisados.

Valor, pois quanto maior for a riqueza dos dados, maior é a importância de saber formular as perguntas corretas no início do processo de análise dos mesmos. É essencial haver um foco no sentido do processo de *Big Data* proporcionar benefícios tangíveis para o negócio.

[1]Fonte: "United Nations Economic Commission for Europe ", https://www.unece.org/info/ece-homepage.html

Presentemente as fábricas estão a adquirir sensores para otimizar as suas linhas de produção, analisando-as e ajustando os processos de acordo com os resultados em tempo real; médicos estão a monitorizar pacientes 24 horas para poderem utilizar a informação recolhida a longo prazo, no sentido de diagnosticar e prever doenças com mais exatidão; empresas estão a recolher informação sobre nós, tais como o nosso nome, o nosso número de telefone, onde vivemos, os nossos hábitos de compras online, e em muitos casos, o que estamos interessados em comprar, por vezes até antes de nós próprios.

O *Big Data* é uma nova forma de ver o mundo, de usar estatísticas e de tomar decisões de negócio. A importância deste conceito do século XXI é imenso, pois a análise de uma grande quantidade de dados, através das técnicas existentes do mesmo, pode levar por parte das empresas a uma redução de custos e de tempo, a desenvolvimento de novos produtos e ofertas otimizadas, e acima de tudo, a decisões mais inteligentes e com um grau menor de risco de um modo geral. Se combinado com a potência das técnicas de *analytics*, podem ser realizadas tarefas como determinar a causa de falhas, problemas e defeitos em quase tempo real, gerar ofertas de acordo com os padrões de compra dos utilizadores ou detetar comportamentos fraudulentos.

Entre o final do século XVII e o princípio do XVIII tivemos a revolução agrícola, seguida da revolução industrial no século XIX, e estamos perante, na minha opinião pessoal, a revolução da Internet e dos dados, e poder participar ativamente na mesma é uma enorme motivação e responsabilidade.

Estamos apenas no início do *Big Data* e não sabemos como poderá mudar tudo. De forma a se poder capturar todo o valor da tecnologia e lidar com toda esta nova geração de dados, surge a necessidade de se conseguir processar e armazenar grandes volumes de dados em tempo real, de forma altamente eficiente. Precisamos efetivamente de plataformas e soluções para dar sentido ao *Big Data* e a realização deste trabalho vai objetivamente desde a exploração das mesmas à sua implementação.

1.2 Contexto

Este trabalho situa-se no contexto do processamento de grandes quantidades de dados em tempo real e o cenário da aplicação a ser desenvolvida pertence ao setor dos Intelligent Transportation Systems (ITS).

A área dos transportes inteligentes tem um papel extremamente importante a nível tecnológico e social. Quanto maior o crescimento económico de um país, maior é a procura e pressão sobre os meios de transporte do mesmo, e caso não haja uma estrutura para suportar esta carga, o desenvolvimento do país encontrará maiores desafios e dificilmente se concretizará. A nível tecnológico a utilização de técnicas de *Big Data* para analisar os dados produzidos pelos transportes pode ajudar a compreender o uso que as pessoas fazem dos diversos modos de transporte e responder a perguntas como: Que caminho seguir? Qual o período mais agitado para cada tipo de transporte? Qual o melhor transporte a escolher e a que horário? É necessário reforçar os transportes?

A partir do cruzamento de todas estas informações, disponíveis para serem recolhidas, podem ser tomadas decisões como a melhoria de rotas existentes, otimização de viagens e redução do tempo e custos dos passageiros, daí a importância do *Big Data* e das suas técnicas nos transportes.

O avanço tecnológico, principalmente da última década, no setor dos transportes inteligentes proporcionou a profunda mudança de paradigma que estamos prestes a vivenciar. Com temas como veículos sem condutor, voos com uma velocidade supersónica, veículos elétricos, drones de passageiros ou até mesmo o conceito de *mobility as a service*, estamos a criar uma nova geração de formas de mobilidade, e por isso, temas inimagináveis até há bem pouco tempo, como o de *smart cities*, estão cada vez mais perto de se tornar uma realidade. Segundo Tony Seba, responsável pelo estudo *Rethinking Transportation 2020 - 2030* [6], daqui a 8 anos todo o transporte terrestre será movido a energia elétrica, acabando assim por exterminar todas as formas de transporte que usam motores movidos a combustíveis fósseis. Para além disso, as pessoas vão deixar de ter viatura própria, utilizando *drones* e veículos elétricos como um serviço de deslocação sempre que precisem.

Quem irá beneficiar de toda esta revolução nos transportes é a economia global e o ambiente. A utilização de veículos elétricos pode ser até 10 vezes mais barata que os automóveis movidos a gasolina ou a gasóleo, e em termos ambientais são muito mais vantajosos [6]. Como também passará a existir um aumento progressivo de veículos autónomos, graças à tecnologia, os acidentes serão uma raridade, o que irá acabar com as intermináveis filas de trânsito que se registam nos dias de hoje. Por todas estas razões, as populações terão uma melhor experiência de transporte, tanto a nível económico como de comodidade, enquanto que ao mesmo tempo contribuem para um desenvolvimento mais sustentável do mundo onde vivem.

1.2.1 Contextualização do Problema

O foco deste trabalho, como já referido, são os sistemas de transporte inteligentes (ITS). O tráfego e os transportes desempenham um papel muito importante na economia atual. Com o avanço da tecnologia vivemos num mundo onde a conectividade é perfeita, e por isso a crescente investigação e investimento neste setor. É de realçar ainda a dependência dos transportes por parte das pessoas nos últimos anos, e a crescente migração das mesmas para as grandes metrópoles [10], o que levou ao aumento de problemas de mobilidade, principalmente nas grandes cidades.

Atualmente, o congestionamento rodoviário é um problema que afeta um número elevado de pessoas, essencialmente aquelas que habitam em grandes áreas metropolitanas, é estimado que em média 40% das pessoas passa pelo menos uma hora na estrada por dia [51]. Esta congestão de tráfego afeta negativamente a qualidade de vida, diminuindo a produtividade pessoal e empresarial, aumentando a quantidade de CO2 que pode ter impactos negativos na saúde, criando poluição sonora, desperdiçando milhões de litros de combustível por ano e dificultando a implementação de uma rede de transportes públicos. Uma grande parte deste congestionamento não é causado pela falta de capacidade por parte da rodovia, mas sim por acidentes e os destroços por eles causados (60%) [10]. Portanto, a redução dos mesmos e a sua deteção o mais rapidamente possível, pode levar à diminuição do congestionamento rodoviário e aos custos negativos a ele associados, bem como ao aumento da segurança rodoviária.

Alguns destes problemas podem ser resolvidos ao serem implementadas novas políticas de transporte, suportadas essencialmente na tecnologia de que dispomos atualmente. Existem várias abordagens que se podem tomar para a resolução de alguns dos problemas acima enunciados, nomeadamente impôr restrições a alguns veículos com base no seu número de matrícula, como já executado durante os Jogos Olimpícos de Pequim de 2008 [51], ou melhorar as infraestruturas da rodovia, aumentando o número de estradas. Apesar destas estratégias serem bastante válidas, apresentam desvantagens evidentes como o custo elevado ou a impossibilidade de permanência a longo prazo.

Porém, novas abordagens estão a surgir com a mudança de rumo dos ITS, devendo-se esse facto ao desenvolvimento de várias áreas e à adoção de novas tecnologias como sistemas de posicionamento, redes de sensores Wireless, telecomunicações, processamento de dados, ou até mesmo operação virtual [33]. Isto significa que as aplicações de ITS devem ser capazes de detetar grandes quantidades de dados, processá-los e inferir através deles informações úteis de modo a que possam ser tomadas decisões e ações eficientes. Todavia, para haver verdadeiramente uma mudança na forma como nos transportamos, urge a necessidade dos novos sistemas de transporte inteligentes serem capazes de tratar dados de tráfego em tempo real, com o intuito de se poderem tomar decisões mais céleres e eficazes. Exemplos desta necessidade incluem o aviso dos consumidores das estradas em caso de elevado congestionamento, estradas cortadas devido a incêndios, condições climatéricas adversas numa determinada auto-estrada, reordenação do fluxo rodoviário,

entre outros.

É certo que devido ao elevado número de veículos que circula pelas cidades, é quase impossível evitar o engarrafamento, particularmente a certas horas do dia. Ainda assim, as condições podem ser agravadas devido a fatores externos, tais como as condições meteorológicas ou a ocorrência de acidentes e, por conseguinte, a previsão em tempo real das condições de tráfego torna-se bastante interessante e útil.

Na Europa, desde 2001 que várias cidades já adotaram esquemas de pagamento de viaturas, principalmente para combater o congestionamento e/ou problemas ambientais. A maioria das cidades tomou medidas como a adoção de preços constantes durante o dia para a entrada nas grande áreas metropolitanas. Porém, com algumas exceções como o caso de Estocolmo ou de Londres, em que o imposto para entrar ou sair do centro destas duas cidades varia com a hora do dia em que o veículo entra ou sai da área de imposto de congestionamento [35], criando uma espécie de imposto dinâmico. Existem outros exemplos da adoção de Sistemas de Transporte Inteligente como acontece em Singapura, que desenvolveu um plano em 2008 de forma a ter um sistema de transportes completamente integrado até 2020. Esta integração consiste basicamente na utilização do já existente sistema de emissão de bilhetes, de transportes públicos e de estacionamento, integrando tudo num só, o que irá permitir introduzir tarifas de acordo com a distância percorrida num futuro próximo [24]. Além destas medidas e devido ao avanço de várias tecnologias, principalmente da *cloud*, o tópico dos veículos conectados e o investimento nos mesmos por parte das empresas da indústria automóvel tem vindo a aumentar [46]. Os conceitos de Vehicle-to-Vehicle (V2V) e Vehicle-to-Infrastructure (V2I) ganham cada vez mais força, indicando que os carros "comunicarão" entre si, trocando dados, como a emissão de alertas para evitar possíveis colisões. Trocarão dados com sensores localizados em semáforos, paragens de autocarro, casa, escritório, *smartphones*, ou até mesmo com estradas para obter atualizações de tráfego e alertas de congestionamento em tempo real.Estas aplicações são apenas uma pequena parte das que já existem ou que estão a ser desenvolvidas no âmbito dos transportes inteligentes.

Efetivamente os ITS enfrentam não só grandes oportunidades, tais como o consumo inteligente de recursos e o aumento da quantidade de dados que são produzidos, mas também grandes desafios, como a segurança rodoviária, o ambiente, a congestão do tráfego e o volume de dados. É de notar que a quantidade de dados é não só uma oportunidade mas também um desafio dos ITS. De facto, há um problema quando temos de processar fluxos ilimitados de dados em tempo real, de tal forma que levou à criação de uma nova classe de aplicações capazes de lidar com este aspeto.

Tem havido um grande esforço para melhorar os tradicionais DBMS (Database Management Systems), conjunto de softwares responsáveis pela manipulação e recuperação de dados guardados numa base de dados, nos últimos anos. Apesar de oferecerem excelente performance no acesso aos dados, sofrem imenso quando estes têm de ser inseridos de forma contínua. Por esse motivo, vários sistemas de processamento e persistência de dados têm sido desenvolvidos de forma a melhorar a escalabilidade e performance em

relação aos DBMS, porém sempre com a incapacidade de realizar análise contínua aos dados [9]. Algumas novas propostas têm surgido, tal como os DSMS (Data Stream Management Systems), sistemas preparados para lidar com *data streams*, fluxos de dados em tempo real [23]. Além do mais têm a capacidade de realizar consultas permanentes aos dados e desta forma produzir novos resultados à medida que os dados vão chegando. Outra das soluções são os Data Stream Warehouses (DSW) que são uma extensão dos DBMS, mas com a característica de conseguirem inserir novos dados em tempo real [23].

Todas estas soluções de gestão de dados serão aprofundadamente abordadas no Capítulo 2 (Estado da Arte) deste documento, todavia para se ter um panorama geral da comparação entre todos estes sistemas podemos observar a Figura 1.2, abaixo representada.

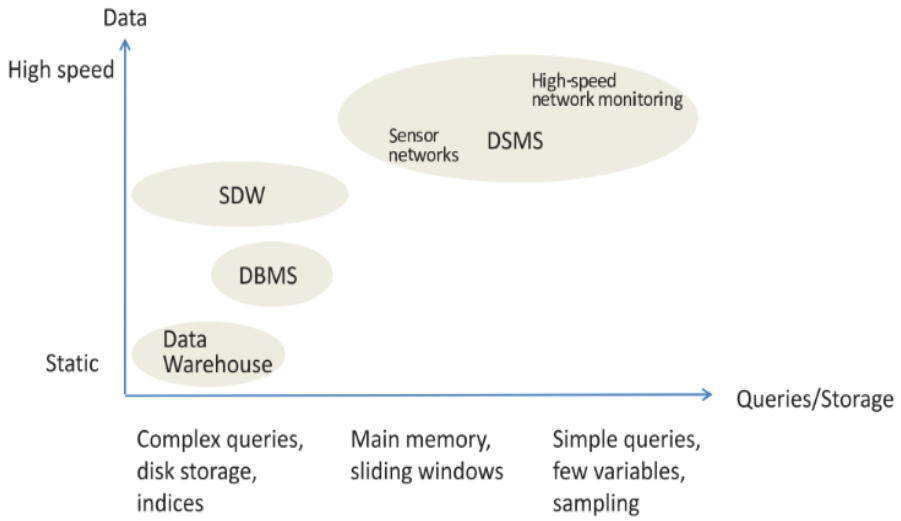


Figura 1.2: Comparação entre os vários Sistemas de Gestão de Dados [2]

[2] Fonte: "Data Stream Management", Autores: Lukasz Golabm, M. Tamer Özsu, p.11

1.3 Problema

Percebido o contexto do trabalho e os inúmeros problemas que ainda afetam o setor dos ITS, o que se propõe neste trabalho é conceber uma aplicação protótipo de *real time analytics* para tentar minimizar alguns deles. Com esse propósito utilizou-se a ferramenta Apache Storm, uma das soluções existentes no que diz respeito a *stream processing*, de forma a ler o valor dos sensores guardados numa base de dados, filtrá-los, processá-los para poder retirar alguma informação dos mesmos, e por fim criar uma visualização em forma de mapa ou gráfico para uma melhor perceção da informação que os dados nos transmitem.

Assim, ao monitorizar-se o trânsito em tempo real podem ser identificadas várias situações como: congestionamento rodoviário, padrões de tráfego, padrões de velocidade, excesso de velocidade, situações atípicas que podem significar acidentes, trabalhos na via, entre várias outras. Ao conseguir identificar-se todas estas situações em quase tempo real podem ser minimizados os impactos das mesmas. A sua análise pode ajudar a tomar decisões tão importantes como a reorganização do trânsito em algumas zonas mais sobrecarregadas, melhorando assim todo o fluxo de viaturas e consequentemente a qualidade da viagem por parte dos utilizadores.

O conjunto de dados, ou *dataset* utilizado, que foi adquirido em formato JSON e só depois exportado para uma base de dados MongoDB, apresenta dados desde Janeiro de 2016 até Maio de 2017 com valores de 349 sensores distintos espalhados por autoestradas, estradas secundárias e estradas principais na Eslovénia. No entanto, estes dados apresentam alguns problemas e diferenças que variam de mês para mês e que podem dificultar o seu posterior processamento.

Os referidos sensores contêm dados como data e hora de medição, ocupação, velocidade, estado do trânsito, localização geográfica, entre outros. Contudo, os mesmos não estão representados no total em todos os meses. Existem meses que começam com um determinado número de sensores, e que algures durante o mês esse número é alterado, isto é, é adicionado ou subtraído um sensor, sendo que o número máximo de sensores representados por mês é de 349. Outra diferença é a taxa a que estes registos de dados são efetuados, existem meses em que a taxa de registo é realizada de 5 em 5 minutos, outros em que esta aumenta para 10 minutos, e até pontualmente existem registos nos quais não é respeitada nenhuma das anteriores.

Como podemos perceber este *dataset* apresenta diferentes variáveis e características que podem representar sérios problemas na sua análise, nomeadamente pouca qualidade e indisponibilidade dos dados, registos duplicados, valores nulos de medição ou metadados errados. No Capítulo 3 deste trabalho podemos encontrar detalhadamente todas as transformações realizadas aos dados, bem como a qualidade e disponibilidade dos mesmos.

Para além destes problemas com o *dataset*, podem surgir problemas relacionados com o facto das tecnologias existentes ainda não estarem suficientemente desenvolvidas para a realização do processamento de dados de forma eficiente, ou com o poder computacional necessário para realizar o processamento em causa.

Em tom de conclusão, a pergunta de investigação que esta tese pretende resolver é como se pode tratar os dados de forma mais eficiente.

1.4 Abordagem

A abordagem seguida por esta tese é uma abordagem tipicamente utilizada por várias empresas e profissionais no ramo das tecnologias de informação para resolver problemas que envolvem mineração de dados com o nome de Cross Industry Standard Process for Data Mining (CRISP-DM) [47]. Esta metodologia é dividida em 6 fases: (1) *Business Understanding* (Entender o Negócio/Problema), (2) *Data Understanding* (Compreensão dos Dados), (3) *Data Preparation* (Preparação dos Dados), (4) *Modelling* (Modelação dos Dados), (5) *Evaluation* (Avaliação) e (6) *Deployment* (Implementação), como ilustrado na Figura 1.3. Esta metodologia apresenta vantagens diferenciadoras, como o facto de ser independente da indústria e da ferramenta de trabalho utilizadas, adaptando-se assim a qualquer caso onde se analisa uma grande quantidade de dados.

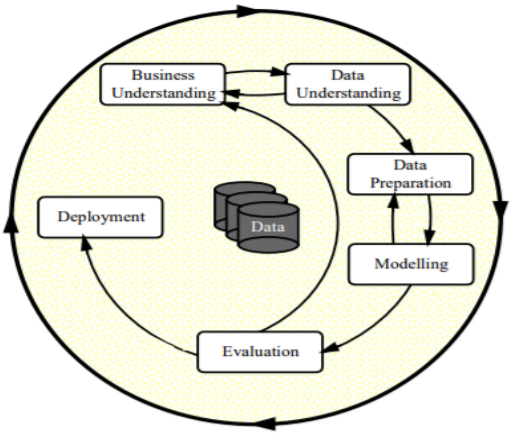


Figura 1.3: Metodologia CRISP-DM para data mining [3]

Para fins ilustrativos, vejamos uma descrição das etapas do método utilizado [etapas (2), (3) e (6)] tendo em conta os objetivos deste trabalho:

- Compreensão dos Dados (Etapa 2): Os dados disponíveis para análise são bastante heterogéneos e contêm informações tais como: a localização dos sensores, latitude e longitude, velocidade, nome da rodovia, data e hora, e número de veículos que passam num determinado período de tempo, representando então dados na ordem

[3] Fonte: "CRISP-DM: Towards a Standard Process Model for Data Mining", Autores: Rüdiger Wirth, Jochen Hipp, pag.5

de grandeza dos *Gigabytes* para serem analisados. Os dados são de vários tipos como booleanos, strings e numéricos. A nível da quantidade de dados, temos os suficientes para poder chegar a conclusões ou previsões bastante precisas e todos eles apresentam características importantes para a resolução do problema. Embora a grande quantidade de dados seja vantajosa no que diz respeito à precisão dos resultados, pode afetar o sistema em termos de tempo de processamento, e por isso possivelmente seja necessário apenas analisar um subconjunto de todos esses dados. Resumidamente, é interessante relacionar alguns destes parâmetros para poder retirar algumas informações, como associar o número de automóveis nas autoestradas durante a época do ano (e.g.: Inverno, Primavera, Verão e Outono), ou esse mesmo índice com a velocidade das viaturas a certas horas do dia. Como podemos ver as hipóteses são imensas e será interessante perceber que conhecimento emerge destes relacionamentos.

- Preparação dos Dados (Etapa 3): Todo o grande volume de dados possui algum tipo de deficiências, principalmente os dados que provêm do mundo real. No *dataset* analisado, proveniente de sensores localizados em várias estradas espalhadas por toda a Eslovénia, existem várias incongruências que têm de ser identificadas e eliminadas sob pena de se produzirem resultados não confiáveis. Podemos destacar alguns exemplos como valores nulos de velocidade e ocupação de viaturas, registos completamente divergentes quando relacionados, códigos ou nomes errados, entre outros. Além disso, a preparação dos dados tem um papel preponderante para a próxima fase desta metodologia, a de modelação dos dados, de forma aos mesmos respeitarem os requisitos da plataforma de *streams* em questão, o Apache Storm.

- Implementação (Etapa 6): Nesta fase recorreu-se ao conhecimento adquirido na utilização da metodologia para a apresentar de maneira a que o utilizador comum, sem qualquer conhecimento informático, a possa utilizar. No caso específico desta tese é através de uma página Web que todo este conhecimento, quer na forma de gráficos e/ou mapas, fica disponível para o entendimento dos utilizadores da aplicação. É de sublinhar que as várias formas de visualização em tempo real da aplicação podem ser vistas em simultâneo e igualmente em tempo real em várias páginas Web, contribuindo assim para a simulação do que poderia ser uma aplicação de monitorização de tráfego em tempo real.

O próximo Capítulo (Estado da Arte) tem um subcapítulo dedicado inteiramente a esta metodologia de trabalho, utilizada para analisar grandes volumes de dados e onde serão descritas detalhadamente todas as fases da mesma.

1.4.1 Contribuições

Este projeto situa-se no âmbito do projeto piloto OPTIMUM [30], um projeto financiado pela União Europeia e cujo objetivo principal é divulgar soluções de tecnologias de informação de ponta para melhorar o trânsito, o transporte de mercadorias e a conectividade do tráfego em toda a Europa. O consórcio do projeto é constituído por 18 parceiros de 8 países diferentes, entre os quais se incluem a TIS – Transportes, Inovação e Sistemas, a IP – Infraestruturas de Portugal (gestora da rede rodo-ferroviária nacional), a Luís Simões e a UNINOVA.

Todo os dados utilizados provenientes da Eslovénia para o desenvolvimento deste trabalho foram disponibilizados pelo projeto, mais concretamente por um dos parceiros do mesmo. Por este motivo, todo o módulo de processamento em tempo real proposto neste trabalho irá contribuir para o projeto OPTIMUM. Para além deste contributo, o trabalho realizado nesta tese contribui de várias outras formas, das quais se destacam nomeadamente:

- Harmonização e limpeza dos dados dos sensores do projeto;
- Análise dos níveis de qualidade e disponibilidade dos dados;
- Análise dos padrões de utilização de um conjunto de autoestradas, estradas principais e estradas secundárias;
- Proposta de arquitetura que implementa o processamento de dados de tráfego de forma paralela;
- Proposta de uma arquitetura de comunicação entre o processamento e a visualização dos dados;
- Implementação de um protótipo que instancia as arquiteturas referidas.
- Contribuição no desenvolvimento do artigo científico “User Interface Support for a Big ETL Data Processing Pipeline: An application scenario on highway toll charging models” [13]

1.5 Estrutura do Documento

Esta tese encontra-se organizada e distribuída pelo seguintes capítulos:

- Capítulo 1: Introdução acerca do que é o *Big Data* e de como surgiu este paradigma. Apresenta a declaração do problema relacionado com o setor dos transportes inteligentes e a abordagem seguida para propôr uma solução para o mesmo.
- Capítulo 2: Apresenta os principais elementos estudados no âmbito do trabalho. Uma primeira secção mais dedicada a sistemas de gestão de *streams* e uma segunda secção com mais enfoque sobre as ferramentas de *stream processing* existentes, particularmente sobre a ferramenta utilizada neste trabalho, o Apache Storm.
- Capítulo 3: Este capítulo faz uma introspeção profunda sobre toda a fonte de dados utilizada no trabalho. Desde a descrição, disponibilidade e qualidade dos dados, à sua seleção, limpeza e exploração.
- Capítulo 4: Apresenta uma visão sobre as várias tecnologias utilizadas na implementação do protótipo, bem como a sua arquitetura. Evidencia-se a arquitetura de todas as topologias desenvolvidas na ferramenta de processamento de *streams*, o Apache Storm. Para além de tudo isto, o último subcapítulo representa uma visão de como os dados fluem desde o início, na base de dados, até à visualização, no browser.
- Capítulo 5: Este capítulo descreve os testes realizados às topologias do Apache Storm desenvolvidas ao nível de performance e escalabilidade, ao serem executadas em *cluster*. São também demonstrados os resultados das várias visualizações desenvolvidas. Por fim, é realizada uma discussão de todos os resultados obtidos.
- Capítulo 6: O capítulo final faz uma analogia entre o que foi estudado e o que foi implementado, comparando o que foi proposto e obtido. É também apresentada uma reflexão do que poderia ser melhorado no futuro.

Capítulo 2

Estado da Arte

Este capítulo apresenta toda a pesquisa efetuada e é a base para a implementação da aplicação desenvolvida. Aborda os principais temas de pesquisa na área do processamento de *streaming* de dados, bem como as suas principais ferramentas, e na área de mineração de dados. É de sublinhar a análise detalhada da ferramenta de processamento em tempo real utilizada neste trabalho, o Apache Storm, bem como a sua comparação com outras existentes que têm a mesma finalidade.

2.1 Técnicas/Tecnologias de *Data Processing*

Ao longo dos últimos cinco anos tem havido o aparecimento de várias framework *open source* para tratar grandes volumes de dados, como o *Hadoop*, que suportada no seu conceito de Map Reduce, foi uma das primeiras plataformas a dar sentido ao *Big Data*, e até à atualidade a mais utilizada para resolver problemas de processamento de grandes quantidades de dados [39]. O *Hadoop* tem como objetivo a análise de grandes conjuntos de dados estáticos, que são recolhidos ao longo de um período de tempo e por esse motivo é classificada como sendo uma plataforma de processamento em lotes (*batches*).

No entanto, recentemente, observou-se que o volume não é o único desafio do grande processamento de dados, e que a velocidade desse processamento é também um dos mais importantes neste novo paradigma da sociedade. Na verdade, para se processar um *dataset* que está constantemente em mudança, não é viável aplicar o método tradicional de armazenar e analisar *a posteriori*. A razão para isso é óbvia: em primeiro lugar, uma grande quantidade de dados não é fácil de gerir; e em segundo lugar, no momento em que se inicia a análise aos dados, estes podem já ter perdido o seu valor. Uma vez que os dados precisam de ser processados em tempo real, a latência no processamento dos mesmos tem de ser bastante reduzida quando comparada com sistemas de processamentos em lote

(*batch*), como o *Hadoop*.

A limitação da abordagem descrita anteriormente levou ao desenvolvimento de técnicas e tecnologias distribuídas, para o processamento em tempo real, novas e sofisticadas. O processamento de dados tendo como preocupação principal a latência é definido como processamento de fluxo de dados, ou em inglês, *stream processing*. A pesquisa na área de processamento de fluxo de dados pode ser dividida em três áreas [52]:

- Sistemas de gestão de fluxo de dados, onde as linguagens de *online query* são exploradas (DSMS);
- Algoritmos de processamento de dados online cujo objetivo é processar os dados numa única passagem;
- Plataformas de *stream processing* que permitem a implementação e escalabilidade de aplicativos baseados em processamento de fluxos de dados.

No âmbito deste trabalho serão abordadas somente a primeira e a terceira área de pesquisa de processamento de fluxo de dados.

2.1.1 Data Stream Management Systems

Inúmeras aplicações processam grandes volumes de streaming de dados. Exemplos incluem uma ampla variedade de dados, como arquivos de *logs* gerados por clientes ao utilizarem seus aplicativos móveis ou da Web, compras de *e-commerce*, redes de sensores, informações de redes sociais, transações de cartões de crédito, entre outros.

Um fluxo de dados é um conjunto de dados que é produzido incrementalmente ao longo do tempo, ao invés de estar disponível na íntegra antes do seu processamento começar. Obviamente, dados completamente estáticos não são práticos e até mesmo bases de dados tradicionais podem ser atualizados ao longo do tempo. No entanto, novos problemas surgem ao processar fluxos ilimitados em quase tempo real. De facto, não é possível executar operações complexas em fluxos de alta velocidade ou manter a transmissão de *Terabytes* de dados brutos para um sistema de gestão de dados [15]. Por essa mesma razão houve uma evolução das base de dados tradicionais para poderem atender a este problema.

Aplicações de monitorização como aquelas enunciadas no início deste texto são muito complicadas de implementar com as tradicionais DBMS. A razão principal é o facto de estas serem bases de dados sequenciais. O seu modelo básico de computação está errado para responder aos desafios das aplicações, pois é um modelo Human-Active Database-Passive (HADP) [28]. Neste modelo, os seres humanos modificam ativamente o conjunto de dados, ou seja, tudo o que acontece no banco de dados é explicitamente realizado pelo utilizador/programa, havendo então a impossibilidade da base de dados enviar notificações quando certas situações específicas são detetadas. Geralmente, em situações onde temos de lidar com dados em *stream* há a necessidade de usar um modelo diferente

de interação, o modelo Database-Active Human-Passive (DAHP), que automaticamente atualiza resultados e notifica ativamente os utilizadores das alterações. As soluções que despoletaram pertencem a este conjunto de "bases de dados ativas" e serão analisadas de seguida.

Surgiram algumas novas soluções como os DSMS (Data Stream Management Systems) ou os DSW (Data Stream Warehouses). Os DSMS são recentes sistemas de gestão de dados capazes de lidar com *data streams*, fluxo de dados em tempo real, contínuos, ordenados (implicitamente por hora) sendo impossível controlar a ordem pela qual chegam [23], e com tabelas de dados. Estes sistemas conseguem conectar-se a uma ou mais fontes de *streams*, e são capazes de processar *queries* contínuas aos dados, ou seja, que permanecem infinitamente, enquanto os dados em fluxo contínuo são transitórios. A principal característica é que os dados produzidos por *streams* não são armazenados, mas sim processados "*on the fly*". Esta última definição é exatamente o oposto das bases de dados padrão (DBMS), onde os dados são permanentes e as *queries* transitórias. Por estas razões podemos afirmar que a diferença fundamental entre os DBMS clássicos e os DSMS é o seu modelo de fluxo de dados. A Tabela 2.1 sintetiza as principais diferenças entre os dois tipos de sistemas de gestão de dados.

Tabela 2.1: Comparação entre DSMS e DBMS

DBMS	**DSMS**
Persistência de Dados	Dados Transitórios
Streams Finitos	*Streams* "Infinitos"
Acesso Aleatório	Acesso Sequencial
Queries transitórias	*Queries* permanentes
Plano de *queries* fixo	Plano de *queries* adaptável

Como já mencionado, um DSMS fornece um modelo de dados para lidar com tabelas permanentes e com *streams*, permitindo ao utilizador definir *queries* contínuas aos dados. Existem duas abordagens principais para fornecer tal facilidade ao utilizador: a primeira é a definição de uma extensão da linguagem Structured Query Language (SQL), e a segunda é a definição de operadores aplicáveis aos *streams* de dados de forma a produzir o resultado esperado [17].

Como prova da primeira abordagem temos alguns estudos já realizados, como por exemplo, o estudo levado a cabo pela Universidade de Stanford (EUA) que propôs um sistema de gestão de *streams* com o nome de STREAM [5]. Um novo sistema construído de raíz que veio introduzir uma série de novos conceitos e soluções para lidar com data *streams*, particularmente a linguagem Continuous Query Language (CQL) que é uma extensão da linguagem SQL (linguagem padrão das bases de dados relacionais, e.g.: PostgreeSQL), projetada para lidar com *streams* e com relações (conjuntos de *tuples*). A grande diferença é que as consultas realizadas a esta base de dados podem retornar uma relação, com base

numa outra relação, como no SQL, mas também podem retornar um *stream* baseado noutros *streams* e relações, daí ser vista como uma extensão do SQL. Para além disso, esta nova linguagem de consulta introduz o conceito de *sliding window*, onde apenas os últimos N elementos a chegar são considerados para responder a consultas, sendo N o tamanho da janela [8]. Embora esteja disponível um *demo* online para download do software, não é verdadeiramente utilizável na prática [2].

Outro dos exemplos que tomou como opção a adoção de uma extensão da linguagem SQL, foi o Stream Processing Language (SPL) desenvolvido pela IBM. Esta linguagem é utilizada na plataforma de processamento de dados em movimento da empresa, o IBM InfoSphere, e tem como paradigma o processamento de fluxo de dados através de um Directed Acyclic Graph (DAG), gráfico dirigido em que as arestas são fluxos de dados e os vértices são instâncias dos operadores, sendo estes nada mais nada menos do que transformadores de *streams*. O SPL traz várias inovações em relação às restantes linguagens de *streaming*. Para fluidez e reutilização do código, o SPL fornece uma interface de geração de código para C++ e Java. De modo a facilitar a escrita de aplicações bem estruturadas e concisas, fornece operadores compostos capazes de modelar sub-gráficos de *streams* [18]. Estes operadores têm funções como: comunicação com fontes de dados externas; conversão de *streams* num conjunto de *tuples*; manipulações a nível dos mesmos como filtragem, projeção e agregação; correlação de dois *tuples*; imposição de uma ordem na qual os *tuples* são enviados; e ainda a capacidade de definir o intervalo de tempo no qual os dados chegam ao DAG. Operadores sem fluxo de entrada são fontes de dados, tendo o seu próprio segmento de controlo, a maioria dos outros operadores dispara quando há pelo menos um item de dados num dos seus fluxos de entrada, ou seja, executa um excerto de código para lidar com esse item. Apesar de apenas consumir um item de dados de cada vez, o operador pode enviar vários itens no seu fluxo de saída.

Em relação à segunda abordagem, temos o exemplo do sistema desenvolvido na Brown University (EUA), no Massachusetts Institute of Technology (MIT) (EUA) e na Brandeis University (EUA) denominado por Aurora [1]. O Aurora é um sistema de gestão de *streams* com um protótipo totalmente funcional e que inclui um ambiente de desenvolvimento gráfico. O processamento do modelo do Aurora representa essencialmente uma base DAG, tal como o do InfoSphere, da IBM. Os seus principais componentes são o *scheduler*, cuja função é decidir que operadores executar e em que ordem; o *storage manager*, projetado para armazenar filas ordenadas de tuples; e o *load shedder*, cuja função é lidar com situações de sobrecarga (Figura 2.1).

As consultas aos dados neste sistema são feitas através da linguagem Stream Query Algebra (SQuAL), semelhante ao SQL mas com recursos adicionais como *windowed queries*, *queries* que são reavaliadas cada vez que o tamanho da *window de tuples* definida passa. As consultas podem ser executadas ao fluxo de dados ou aos dados históricos armazenados no Aurora através do *storage manager*.

[1] Fonte: "Aurora: a new model and architecture for data stream management", Autores: Abadi, Daniel

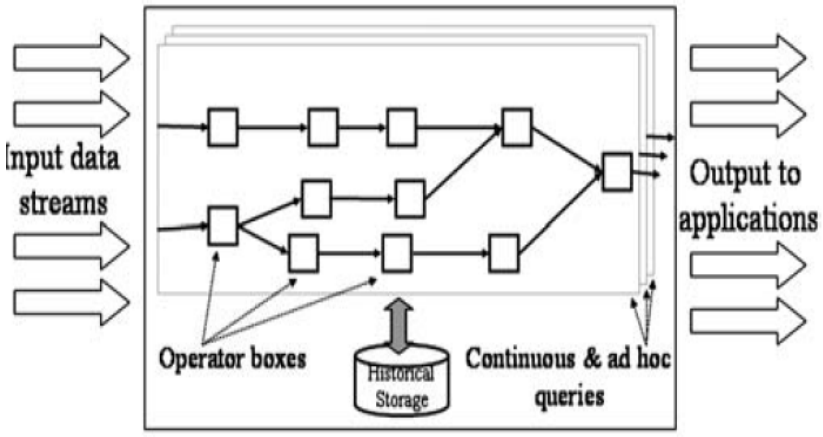


Figura 2.1: Modelo Sistema Aurora[1]

A outra solução que surgiu como alternativa ao DBMS foram os DSW que combinam a resposta em tempo real de um DSMS, ao tentar carregar e propagar novos dados assim que os mesmos chegam, com a capacidade de um *data warehouse* de gerenciar *Terabytes* de dados históricos em armazenamento secundário [17]. Em monitorização de redes como de sensores, por exemplo, tem a capacidade de armazenar fluxos de dados que foram pré-agregados ou pré-processados por um DSMS. O DBStream [9], um novo sistema de monitorização de tráfego online é um exemplo de um DSW. Este sistema importa e processa os dados em pequenos *batches* (lotes de dados). Por um lado lembra um sistema DSMS, no sentido em que os dados podem ser processados rapidamente, por outro os DBMS, pela capacidade de reproduzir *streams* a partir de dados passados. Deste sistema saltam à vista duas características principais: primeiro, suporta *queries* incrementais, isto é, *queries* que atualizam os resultados através de dados previamente gerados, em vez de serem de novo gerados do zero; segundo, em contraste com muitas extensões de bases de dados, este sistema não altera o mecanismo de processamento de *queries*. Por todos estes motivos, este sistema é utilizado amplamente em aplicações de Network Traffic Monitoring and Analysis (NTMA), para funções como deteções de anomalias.

2.1.2 Sistemas Distribuídos de *Stream Processing*

As plataformas de *stream processing* são projetadas para funcionar em tecnologias de computação distribuídas e paralelas, como num *cluster*, de forma a alcançarem o processamento em tempo real de um grande volume de dados. O *stream* (fluxo de dados) é o elemento central destas plataformas e refere-se ao fluxo de dados infinito que existe dentro de todo o sistema. Outro conceito importante nestas plataforma é o de *tuple*, que pode ser visto como partições de um *stream*, ou seja, um conjunto de *tuples* forma um

[1] J and Carney, Don, Ugur and Cherniack, Mitch and Convey, Christian and Lee, Sangdon and Stonebraker, Michael and Tatbul, Nesime and Zdonik, Stan , pag.2

stream. Antes de passarmos concretamente para as plataformas que existem neste contexto, vejamos o modelo geral e os requisitos das mesmas.

2.1.2.1 Modelo Geral

Uma típica plataforma de processamento de *streams* recebe dados através de fontes externas, como por exemplo, Facebook, Twitter e bases de dados, e entrega os resultados do processamento de volta para serem guardados numa base de dados ou para publicação dos mesmos num sistema online, como um website. Desta forma os sistemas de processamento de *streams* baseiam-se essencialmente em dois conceitos fundamentais: fonte de dados externa, capaz de injetar dados para o sistema, e nós de processamento, onde atividades de transformação aos dados são realizadas. Um modelo geral destes sistemas pode ser visto na Figura 2.2.

Os *tuples* de um *stream* são injetados através dos nós de geração (S1, S2) para as unidades de processamento (P1, P2, ...), estas por sua vez consomem-nos, aplicam alguma transformação, e emitem *tuples* para as fases seguintes. Cada unidade de processamento pode decidir não emitir nada, emitir um *tuple* de uma forma periódica, ou até emitir *tuples* após um certo número dos mesmos recebido. A comunicação entre as unidades de processamento é realizada através de sistemas de *push-pull messaging*. A distribuição destes nós e a forma como comunicam é um pouco diferente nos vários sistemas de *process streaming* existentes.

Quando estas unidades conectadas na forma de um DAG são executadas num *cluster*, cada unidade é distribuída paralelamente entre os nós de computação de forma a alcançar alto rendimento. Na forma como esta distribuição é realizada podemos distinguir dois tipos de sistemas: no primeiro tipo, existe um servidor central que orquestra a distribuição das unidades de processamento para os diferentes nós físicos do *cluster*, sendo também responsável pelo balanço das tarefas pelos mesmos; no outro tipo, não existe nenhum nó responsável pela distribuição, funcionando como sistemas distribuídos *peer-to-peer*, dividindo o trabalho igualmente [21].

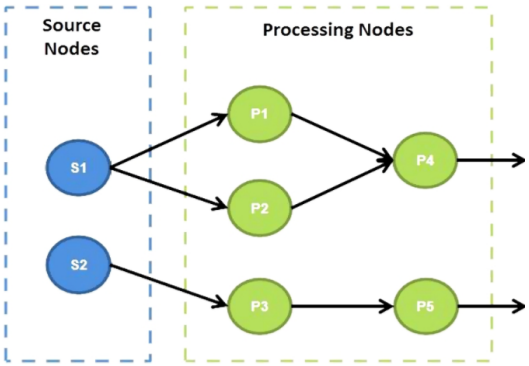


Figura 2.2: Modelo geral de fluxo de dados de uma plataforma de *stream processing*[2]

[2]Fonte: "A Taxonomy and Survey of Stream Processing Systems", Autores: X. Zhao, S. Garg, C. Queiroz

2.1.2.2 Requisitos de um Sistema de *Stream Processing*

Os requisitos de um bom framework de processamento de *streams* assentam fundamentalmente em duas características: latência e escalabilidade do sistema. Michael Stonebraker [42] propôs um conjunto de requisitos que devem estar presentes nestes sistemas, listados abaixo:

- Manter os dados em movimento: Manter a latência no mínimo possível, processando os dados assim que capturados;
- Capacidade para lidar com imperfeições em *streams*: Deter características para lidar com a falta ou desordem de *streams* de dados;
- Utilizar SQL em *query* de *streams*: Permitir *queries* SQL em *streams* de dados para construir operadores;
- Gerar resultados previsíveis: Capacidade para fornecer resultados garantidos e previsíveis;
- Integração de dados armazenados e de transmissão: O sistema deve ser capaz de combinar diferentes fluxos de dados de diferentes fontes externas com várias velocidades de inserção;
- Garantia de segurança e disponibilidade dos dados: Capacidade para fornecer recursos de tolerância a falhas de forma a garantir o processamento contínuo dos dados com o mínimo de perdas;
- Partição e escalabilidade automática: Distribuição e processamento de dados entre vários processadores/CPUs;
- Processamento e resposta imediata: Capacidade para obtenção de resposta em tempo real com sobrecarga mínima para o fluxo de dados de alto volume.

2.1.2.3 Tolerância a Falhas

As falhas de computação distribuída podem ocorrer devido a várias razões, nomeadamente falhas de nós e de rede, erros de software e limitações de recursos. Mesmo que individualmente os componentes apresentem uma probabilidade pequena de falha, quando se encontram a trabalhar em conjunto em sistemas com vários nós, a probabilidade de um dos componentes falhar aumenta substancialmente. Estas falhas têm de ser levadas em conta, pois podem levar a consequências graves como a perda de dados. Em sistemas de processamento em *batch*, o atraso na computação devido a uma falha não é crítico, porém tal não acontece em sistemas de processamento de *streams* onde a latência é uma das principais preocupações. Deste modo, estes sistemas têm de ser capazes de recuperar de uma falha com o mínimo de efeito sobre a computação do sistema em geral.

Existem três semânticas distintas de entrega de mensagens: "exatamente uma vez" (*exactly-once*), "até uma vez" (*at-most-once*) e "pelo menos uma vez" (*at-least-once*). As semânticas dizem respeito à garantia que o sistema dá sobre o processamento ou não de dados em caso de falha. Quando ocorre uma falha na transmissão durante o processamento dos dados, pode ocorrer o reenvio dos mesmos, para que nenhuma informação seja perdida, sendo este o funcionamento da semântica *exactly-once*. Na semântica *at-most-once*, a mais simples, na qual não há recuperação de erros e, ou os dados são processados uma vez, ou são perdidos . Por fim, a semântica *at-least-once*, onde a correção do erro é feita de maneira conjunta para um grupo de amostras, dessa maneira caso ocorra erro com alguma dessas amostras, o grupo inteiro é repetido.

2.1.2.4 Ingestão de Dados

O processo de ingestão de dados num sistema de processamento de *streams* é ligeiramente diferente de apenas reunir ou identificar todos os dados a serem processados. A diferença reside na atividade adicional de formatação dos dados. Este processo geralmente envolve a adição e alteração de conteúdo e formato de dados, com o objetivo de serem utilizados corretamente depois. O motivo para estas transformações é que os fluxos de dados apresentam uma grande diversidade nos seus tipos, enquanto as plataformas de processamento de *streams* podem por vezes ser apenas capazes de processar certos tipos, ou seja, os dados têm de se adequar aos requisitos das plataformas e não o contrário. Presentemente, alguns sistemas de mensagens como Apache Kafka, Apache Flume, ZeroMQ, RabbitMQ, entre outros, fornecem este tipo de conversão, e por essa razão são muito utilizados na integração com sistemas de *stream processing*.

O processo de ingestão de dados engloba outra funcionalidade que é a identificação da fonte de dados. No geral, existem dois tipos de fontes de dados: dados armazenados e dados de transmissão. Embora as plataformas de processamento de fluxo sejam projetadas principalmente para o processamento de grandes conjuntos de dados de transmissão, por vezes os dados armazenados são necessários para realizar uma análise mais abrangente aos mesmos.

2.1.3 Ferramentas de *Stream Processing*

Existem diferentes framework voltadas para o processamento de dados em *streaming*. Entre elas podemos destacar o Apache Samza, o Storm e o Spark Streaming, que veremos em detalhe de seguida, principalmente o Apache Storm devido à sua importância para o trabalho.

2.1.3.1 Apache Storm

Como já mencionado, este foi o sistema de processamento de *streams* utilizado para o desenvolvimento do protótipo deste trabalho. O Apache Storm [4] é uma estrutura

computacional distribuída e em tempo real, desenvolvida inicialmente por Nathan Marz e futuramente adquirida e aprimorada pelo Twitter, que torna o processamento ilimitado de fluxos de dados mais simples. O Storm foi projetado para processar grandes quantidades de dados através de um método tolerante a falhas e escalável horizontalmente. Atualmente é uma das ferramentas mais utilizadas em análises de dados em tempo real, pois é fácil de configurar, operar e garante que cada mensagem é processada através da sua topologia numa tolerância à falha de *at-least-once*.

A arquitetura de processamento de dados no Storm baseia-se essencialmente num conjunto de *tuples* a fluir por topologias. O Apache Storm lê o fluxo de dados em tempo real de uma determinada fonte de dados numa extremidade, e passa o mesmo através de uma sequência de pequenas unidades de processamento, emitindo depois as informações processadas na outra extremidade, como pode ser observado na Figura 2.3. É de sublinhar que a última unidade de processamento pode enviar os dados para serem guardados numa base de dados, ou como já referido, enviar o output da topologia após o processamento para o browser.

A topologia do Apache Storm é onde se encontra toda a lógica da aplicação em tempo real e é composta por unidades de processamento denominadas *spouts* e *bolts* conectadas por *stream groupings*. Estas conexões entre os *spouts* e os *bolts*, não são nada mais nada menos, que as várias formas que o Storm tem de particionar os *tuples* ao longo das várias instâncias/nós de processamento na topologia.

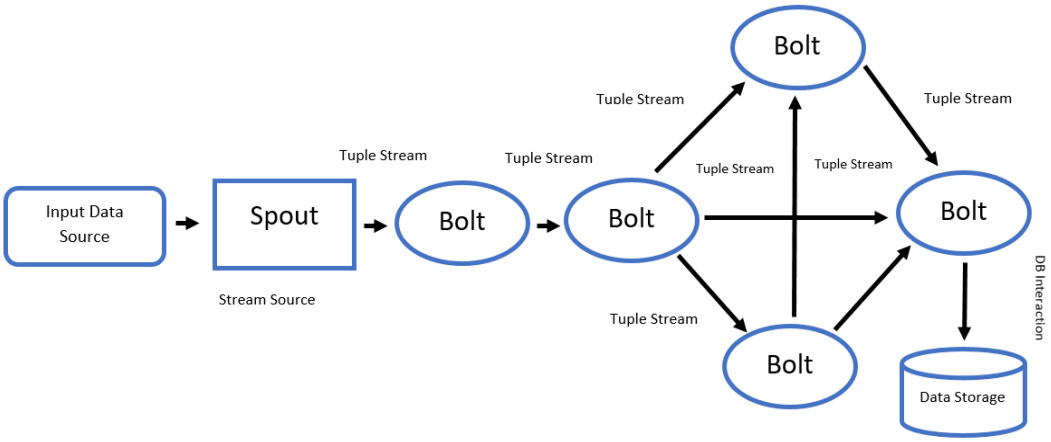


Figura 2.3: Topologia Apache Storm

Existem quatro tipos de *stream groupings* diferentes: o *Shuffle Grouping*, que distribui os *tuples* de forma aleatória para os *bolts* da instância seguinte, garantindo que cada *bolt* recebe o mesmo número de *tuples*; o *Fields Grouping*, em que o fluxo de dados é particionado por um campo específico, ou seja, se o fluxo é agrupado pelo campo "nome" por exemplo, *tuples* com o mesmo campo "nome" irão sempre para a mesma tarefa na instância seguinte; o *All Grouping*, que basicamente replica os *tuples* para a próxima instância de *bolts*; e por fim, o *Global Grouping*, que envia todos os *tuples* para um único *bolt*. Para além de todos estes *stream groupings* que o Storm fornece, também pode ser

implementada uma forma específica de realizar as conexões entre as várias instâncias na topologia por parte do utilizador, cujo nome é *Custom Grouping*.

As definições de todos os constituintes da topologia são as seguintes:

- *Tuple*: Principal estrutura de dados no Storm, lista de elementos ordenados que suporta qualquer tipo de dados;
- *Stream*: Sequência desordenada de *tuples*;
- *Spout*: Fonte de *streams* numa topologia, geralmente leem *tuples* de uma fonte externa de dados, como uma base de dados ou *queues* como Kafka, RabbitMQ ou Kestrel, e que os transmitem para dentro da topologia;
- *Bolt*: Unidades de processamento lógico, que recebem os dados provenientes dos *spouts*, processam-nos e produzem outro output *stream*. Este pode ser transmitido novamente a outro *bolt*, ou pode ser simplesmente armazenado numa base de dados. Têm a capacidade de filtrar, agregar e processar os dados de forma paralela, de forma a acompanhar a elevada taxa de input das aplicações de processamento em tempo real.

De seguida iremos observar a arquitetura do *cluster* do Apache Storm, e entender todos os seus constituintes e funcionamento. Deste modo, iremos perceber como funciona o sistema no seu todo quando submetemos qualquer tipo de topologia no *cluster*.

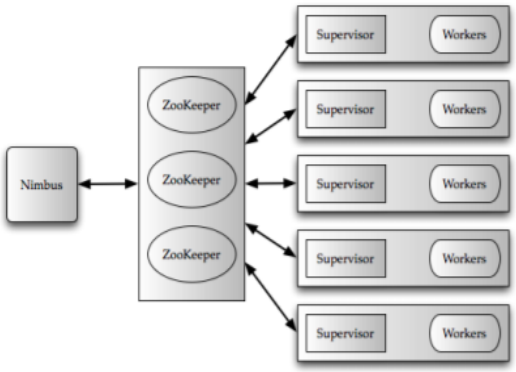


Figura 2.4: Arquitetura Cluster Apache Storm[3]

O *cluster* do Apache Storm possui três diferentes tipos de nós: o Nimbus, o ZooKeeper e o Supervisor, tal como ilustrado na Figura 2.4. O Nimbus é o nó principal do Storm, considerado como *master node*, pois é o responsável pela distribuição e coordenação de toda a execução da topologia. O processamento real da topologia é realizado nos *worker nodes*. Cada *worker* executa uma Java Virtual Machine (JVM), onde são executados os *executors*. Os *executors* são compostos por conjuntos de tarefas que são efetivamente instâncias de

[3]Fonte: Storm Twitter, Autores: Toshniwal, Ankit and Taneja, Siddarth and Shukla, pág.148

bolts e/ou *spouts* da topologia. Desta forma podemos afirmar que o número de tarefas concede paralelismo à topologia e o número de *executors* (*threads*) ao *cluster* [45].

É de enfatizar que associado a cada *spout*/*bolt* encontram-se um conjunto de tarefas a serem executadas em vários *executors* distribuídos por diferentes máquinas (*workers*) num *cluster*, como ilustrado na Figura 2.5.

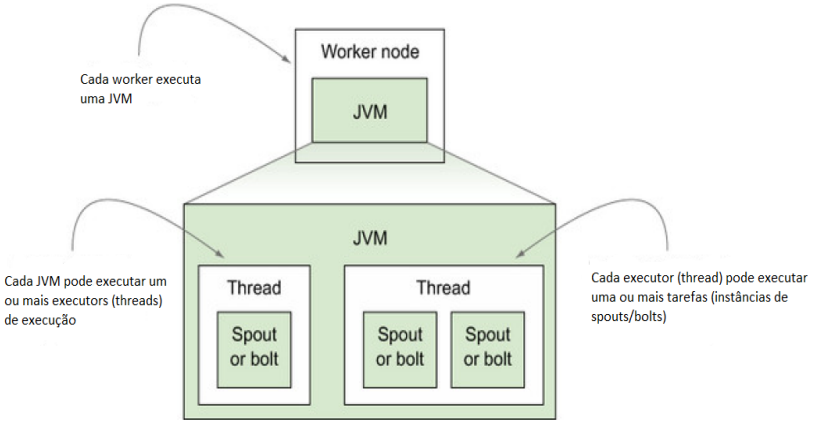


Figura 2.5: Arquitetura dos *worker nodes*[4]

Vejamos internamente como todos os nós interagem e quais as funções dos mesmos ainda não abordados, o ZooKeeper e o Supervisor.

Cada *worker* executa um *daemon* denominado Supervisor, que comunica com o Nimbus e cuja coordenação é realizada pelo ZooKeeper. O Supervisor é executado em cada nó do *cluster* do Storm. Este recebe as informações acerca da quantidade de *workers* na topologia através do Nimbus, e tem como função principal a divisão dos mesmos pelo *cluster*. Para além disso, também monitoriza a "saúde" dos *workers*, e caso haja alguma falha, tem a capacidade de redistribuí-los para outro *worker* presente no *cluster*. Como pode ser observado na Figura 2.6, o Supervisor gera três *threads*. A *main thread* lê a configuração da topologia do Storm, salva o seu estado atual no sistema de ficheiros do sistema e agenda eventos recorrentes de tempos a tempos. Existem três tipos de eventos:

1. ***Hearbeat event***, que está agendado para ser executado a cada 15 segundos na *thread* principal e cuja função é informar o Nimbus do estado dos Supervisors;

2. ***Synchronize Supervisor event***, que é executado de 10 em 10 segundos, na *thread event manager*. Esta *thread* é responsável pela gestão de mudanças na topologia existente. Por exemplo, caso haja adições de novas topologias no *cluster*, esta *thread* faz o download dos ficheiros JAR e automaticamente agenda um *Syncronize Process event* para ser executado.

[4]Fonte: Adaptado de Storm Applied Strategies for real-time event processing, Autores: Sean T.Allen, Matthew Jankowski, Peter Pathirana, pág.91

3. ***Syncronize Process event*** é um evento que ocorre a cada 3 segundos na *thread process manager*. Esta *thread* é responsável por gerir os estados dos *workers*, capta o *hearbeat* dos mesmos a partir do estado atual guardado no sistema de ficheiro pela *thread* principal, classificando-os como *valid*, *timed out*, *not started* ou *disallowed*. Um *worker timed out* indica que este não informou o seu estado durante o intervalo de tempo do evento, sendo assumido como "morto" para o *cluster*. Um *worker not started* indica que o mesmo ainda não foi inicializado, pois pertence a uma nova topologia implementada. Por fim, um *worker disallowed* significa que o próprio não deveria ser executado pois a topologia onde se encontrava foi terminada ou movida para outro nó pelo Nimbus.

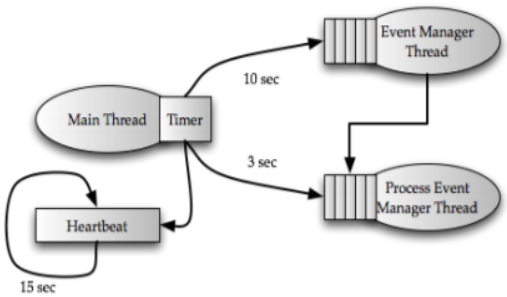


Figura 2.6: Arquitetura *Supervisor Node* - Apache Storm[5]

Os *executors* por sua vez que residem dentro dos *worker* são *threads*, e cada uma destas pode executar várias *tasks*, que são instâncias de *bolts/spouts*. Para mapear *tuples* de entrada e saída, cada *worker* possui duas *threads* dedicadas, isto é, a *worker receive thread* e a *worker send thread*. A *thread* de recebimento escuta um porto TCP/IP e atua como um ponto de *demultiplexing* dos *tuples* de entrada. Confere a tarefa destino de cada *tuple* e coloca-o na *queue* de entrada do respetivo *executor*, onde se encontra a tarefa que lhe é destinada.

Cada *executor* contém também duas *threads*, designadas por *user logic thread* e *executor sent thread*. A primeira *thread* adquire os *tuples* presentes na *in queue*, envia-os para a respetiva tarefa, executa-a, ou seja, executa uma instância de *bolts/spouts* para aquele *tuple*, e coloca o output na fila de saída (*out queue*). Em seguida a *thread* de envio do *executor* agarra nos *tuples* da *out queue* e coloca-os numa *queue* global de transferência. Esta lista contém todos os output *tuples* de vários *executors*. Finalmente, a *thread* de saída de cada *worker* tem a responsabilidade de enviar os *tuples* na *queue* global para as próximas instâncias da topologia. O fluxo explicado anteriormente pode ser então observado na Figura 2.7.

[5]Fonte: Storm Twitter, Autores: Toshniwal, Ankit and Taneja, Siddarth and Shukla, pág. 150

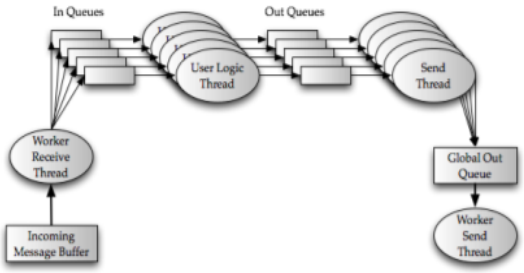


Figura 2.7: Fluxo das mensagens no *worker node* - Apache Storm[6]

Por último, vejamos então de que forma é feito o *deployment* de topologias para o *cluster*. O Nimbus é um serviço Apache Thrift e as topologias do Storm são objetos do mesmo. Esta framework combina uma série de ferramentas de geração de código para criação de serviços que funcionam de forma eficiente e transparente entre várias linguagens de programação, tais como Java, C++, Python, Ruby, Erlang, JavaScript, Perl, entre outras. Por esta razão, as topologias no Storm podem ser implementadas em todas as linguagens de programação. Como parte de submissão da topologia, o utilizador faz o upload dos ficheiros JAR para o Nimbus. O Nimbus por sua vez utiliza o disco local e o ZooKeeper para guardar o estado da topologia, mais concretamente o código do utilizador é guardado no disco local e os objetos Thrift no ZooKeeper. Seguidamente os Supervisors, como já visto anteriormente, comunicam com o Nimbus através de um *heartbeat* periódico, para lhe informar que topologias estão a ser executadas. É de realçar novamente que a coordenação entre estes dois nós é realizada pelo ZooKeeper, que mantém o estado dos *daemons* dos dois nós guardados e atualizado periodicamente. Sendo que esta constitui uma das características mais importantes no que diz respeito à tolerância a falhas deste sistema de *streaming*.

De facto, uma das maiores vantagens da utilização do Apache Storm é este não ter um único ponto de falha [Single Point of Failure (SPOF)] [4]. Podem existir falhas em dois locais, nomeadamente na topologia e nos nós do *cluster*. Em relação ao primeiro local de falhas este é colmatado devido à utilização de um algoritmo de *backup stream*, que resumidamente faz com que os *spouts* retenham as mensagens a ser introduzidas na topologia, até que os *bolts* enviem uma mensagem de confirmação de receção. Se tal não acontecer, os *spouts* voltam a enviar a mensagem, caso contrário, apagam-na da sua fila de saída. Este algoritmo só é possível, pois o Storm atribui um *id* aleatório de 64 bits a cada *tuple* que flui no sistema.

No que diz respeito às falhas nos nós do *cluster*, estas são tratadas pelo Nimbus. Como já explicado, os Supervisors enviam o seu estado ao Nimbus periodicamente e este é guardado pelo ZooKeeper. Caso o Nimbus não os receba, este assume que os Supervisors não estão mais ativos e move os *workers* para outro Supervisor. Desta forma, e sabendo o ZooKeeper do último estado de não ativo do Supervisor, caso este mude o seu estado, o

[6]Fonte: Storm Twitter, Autores: Toshniwal, Ankit and Taneja, Siddarth and Shukla, pág. 149

Nimbus sabe que este voltou a estar ativo e pode novamente enviar para esse nó outros *workers*, reduzindo assim a latência em caso de falha. Este *design* faz com que o sistema seja bastante robusto em relação a falhas que possam ocorrer.

2.1.3.2 Spark Streaming

O Spark, um projeto iniciado pela Universidade de Berkeley (EUA), é uma plataforma para processamento de dados distribuída em tempo real. O Spark suporta diferentes bibliotecas para o processamento de fluxos de dados, entre elas encontra-se o Spark Streaming [40].

Este tipo de processamento de *streams*, ao contrário do Apache Storm, não processa *streams* um de cada vez, em vez disso agrupa-os em pequenos *batches* por intervalo de tempo antes de processá-los. A abstração deste sistema que representa um *stream* de dados, é denominada por *DStream* (*Discretized Stream*). Uma *DStream* é um *micro-batch* de Resilient Distributed Dataset (RDD), sendo estes conjuntos de elementos particionados nos vários nós do *cluster* e que podem operar em paralelo, como ilustrado na Figura 2.8.

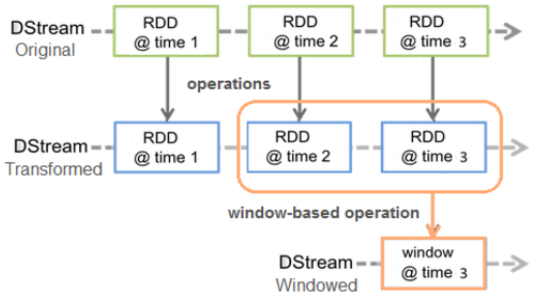


Figura 2.8: Modelo DStream *Discretized Streams* - Spark Streaming[7]

O fluxo de dados é a entrada para o Spark Streaming, que cria os micro-lotes em forma de RDD's, como já referido. Por sua vez, esses *batches* são passados para o Spark convencional, que tem como função a realização do processamento. Um trabalho (*Job*) no Spark é definido como uma computação paralela que consiste em múltiplas tarefas, e uma tarefa é uma unidade de trabalho que é enviada ao *executor*. Podemos observar a arquitetura geral do *cluster* do Spark Streaming na Figura 2.9 ilustrada seguidamente.

[7] Fonte: Website Spark Streaming https://spark.apache.org/streaming/

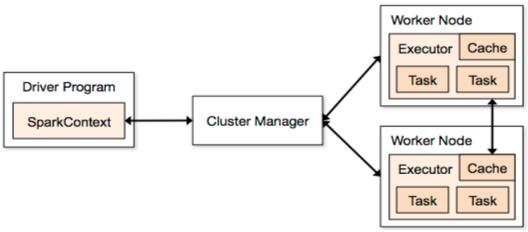


Figura 2.9: Arquitetura geral cluster Spark Streaming [8]

2.1.3.3 Apache Samza

A abordagem do Apache Samza [3] em relação ao modelo de processamento é tal e qual como o do Storm, um *stream* de cada vez. Ao contrário dos outros dois sistemas de processamento de *streams* já abordados, a abstração de *stream* no Samza não é o *tuple* nem o *DStream*, mas sim uma *message*.

Os *streams* são divididos em várias partições e cada uma delas é uma sequência de mensagens ordenadas, exclusivamente de leitura, como ilustrado na Figura 2.10. Este sistema também suporta o processamento em *batch*, consumindo várias mensagens da mesma partição em sequência.

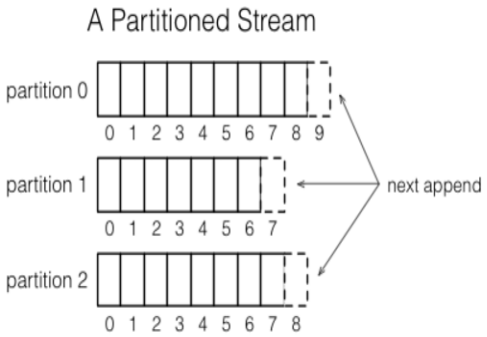


Figura 2.10: Partition Stream - Samza[9]

Pela descrição destes 3 modelos de processamento de *streams*, podemos perceber que as opções para o modelo de processamento se dividem essencialmente em duas: *real time* e *micro-batch*. Estas duas opções têm um grande impacto no que diz respeito à possibilidade de desempenho, que depende essencialmente do caso de uso específico. No caso do *Spark Streaming* em concreto, que realiza processamento em *micro-batch*, este fator faz com que a latência do processo de processamento seja na ordem dos segundos. Sendo o requisito principal da aplicação desenvolvida a velocidade, esta ferramenta nunca seria equacionada. Para além disso, o Storm é de longe a ferramenta com mais compatibilidade a nível de linguagens de programação, podendo ser implementada em qualquer uma, e aquela que tem a maior comunidade de programadores entre as três estudadas [25]. Por

[8] Fonte: Website Spark Streaming https://spark.apache.org/streaming/
[9] Fonte: Website Apache Samza http://samza.apache.org/

todas estas razões, a ferramenta de trabalho escolhida para o protótipo desenvolvido para processar os dados em tempo real foi o Apache Storm.

De forma a obtermos uma visão mais abrangente das diferenças fundamentais entre as três principais plataformas de processamento de *streams* em tempo real, vejamos a Tabela 2.2.

Tabela 2.2: Apache Storm vs Samza vs Spark Streaming

	Storm	**Samza**	**Spark Streaming**
Modelo de Processamento	Um registo de cada vez	Um registo de cada vez	micro-batch
Latência	milisegundos	milisegundos	segundos
Taxa de Output	10K+ por nó por segundo	100K+ por nó por segundo	100K+ por nó por segundo
Tipo de garantia de processamento	pelo menos uma vez	pelo menos uma vez; suporta exatamente uma vez	exatamente uma vez

2.2 CRISP-DM

Como já mencionado, a metodologia que se revelou mais adequada para a realização de uma completa introspeção sobre os dados foi a metodologia CRISP-DM, que é frequentemente utilizada em problemas que envolvem data mining.

Antes de mais, é fulcral entender o verdadeiro significado de data mining e como surgiu este modelo padrão. O conceito data mining nasceu da interseção de três áreas distintas: estatística, inteligência artificial e *machine learning*. Este conceito é parte de um processo maior conhecido como Knowledge Discovery in Database (KDD), cuja definição no livro de Usama M. Fayyad [12] é "o processo não trivial de identificação de padrões válidos, desconhecidos, potencialmente úteis e, no final das contas, compreensíveis em dados", com o qual é tantas vezes confundido. Este processo é composto por várias fases, como ilustrado na Figura 2.11, que são executadas de forma dinâmica, isto é, sem uma ordem implícita.

O processo do KDD inicia-se com o conhecimento do domínio da aplicação e dos objetivos a serem alcançados. Seguidamente é realizada uma seleção a todos os dados que irão ser alvo de descoberta. Como a qualidade dos dados é fundamental para a obtenção de resultados fidedignos, a limpeza aos mesmos e a sua compreensão são muito importantes para o sucesso do data mining, fase do pré-processamento. É de notar que esta fase pode ocupar até 80% do tempo necessário para todo o processo, principalmente em casos onde existem bases de dados muito heterogéneas [22].

Feito o pré-processamento, passa-se ainda por outra transformação que é o armazenamento adequado dos dados em *Data Warehouses*, com o intuito de facilitar o uso das técnicas de data mining. Avançando no processo do KDD chega-se então à fase do data mining, cujo objetivo principal é utilizar técnicas/algoritmos de mineração sobre os dados pré-processados com o propósito de extrair padrões ou conhecimentos relevantes sobre os mesmos.

A última etapa do processo intitulada de pós-processamento consiste na interpretação dos padrões descobertos. Nesta fase a informação extraída é analisada de acordo com o objetivo proposto inicialmente e se necessário é filtrada a informação a ser posteriormente apresentada, como possíveis ruídos (e.g.: padrões redundantes).

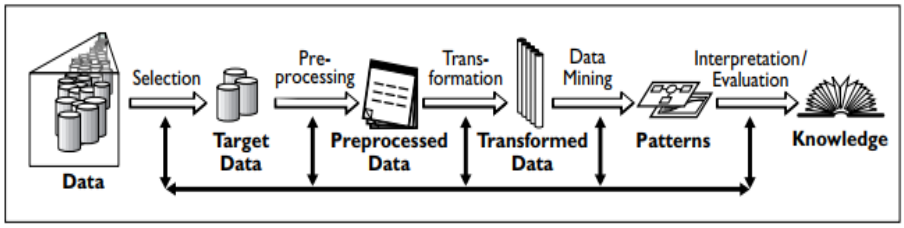


Figura 2.11: Etapas que compõem o KDD[10]

[10]Fonte: "The KDD process for extracting useful knowledge from volumes of data", Autores:Fayyad, Usama and Piatetsky-Shapiro, Gregory and Smyth, Padhraic

Resumidamente, o data mining é um processo criativo que requer um conjunto de habilidades e conhecimentos diferentes. Isto significa que o sucesso ou insucesso de um projeto de data mining está altamente dependente da pessoa ou equipa que se encontra a realizar o mesmo [47]. De facto, o data mining precisa de uma abordagem padrão que ajude a traduzir problemas de negócio em tarefas de data mining, sugerindo transformações e técnicas adequadas aos dados.

Por esta razão foi criado o CRISP-DM [47], um modelo dividido em 6 fases, como ilustrado na Figura 2.12, e que foi a base para a elaboração deste trabalho. É de destacar o sentido das setas entre as fases, que significam que ir e voltar entre as mesmas é um processo sempre vital para o sucesso do modelo, e também as setas do círculo exterior que traduzem a natureza cíclica do processo de data mining.

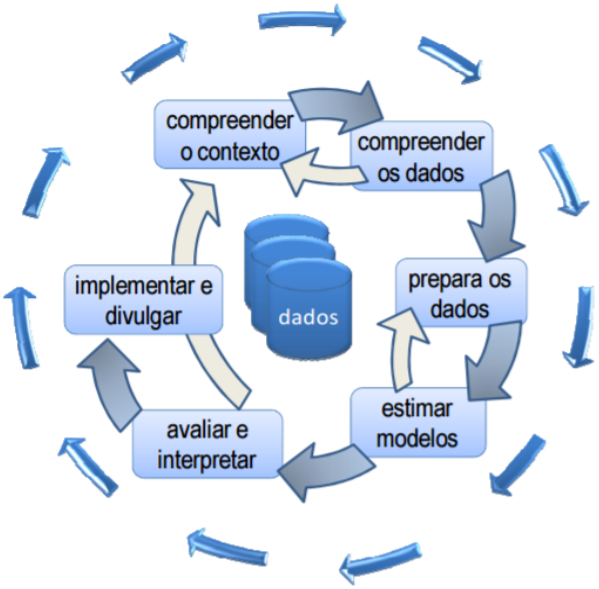


Figura 2.12: Modelo Processual CRISP-DM [11]

1. **Compreender o Contexto (Business Understanding)**

   A fase inicial foca-se em compreender o objetivo e os requisitos do projeto do ponto de vista de negócio, de modo a projetá-los para um problema de data mining. É ainda nesta fase que é traçado um plano preliminar visando atingir os objetivos pretendidos.

2. **Compreender os Dados (Data Understanding)**

   A fase de compreensão dos dados inicia-se com uma recolha de todos os dados disponíveis no contexto da aplicação e que serão objeto de estudo. Prossegue com uma familiarização com os dados através de ações, como a identificação de problemas

[11] Fonte: "Business Intelligence no suporte a decisões sobre comunicações", Autores: Mendes, Armando Brito and Alfaro, Paulo Jorge and Ferreira, Aires, p.25

de qualidade, descrição dos dados ou deteção de subconjuntos interessantes para a formulação de hipóteses de resolução do problema. Como é fácil de perceber, a fase de Business Understanding e Data Understanding estão diretamente conectadas, pois não é possível traçar um plano preliminar sem algum conhecimento sobre os dados disponíveis.

3. **Preparação dos Dados (Data Preparation)**

   A fase de preparação dos dados é uma das mais importantes no que diz respeito ao data mining e a mais morosa também. Numa fase inicial a ideia é limpar os dados, ou seja, excluir campos com dados em branco, excluir características irrelevantes, e corrigir meta-dados que estejam incorretos. Posteriormente, é essencial agregar campos, de forma a poder relacioná-los e das quais possivelmente surgirão novos atributos. Por fim, é necessário a certificação de que os dados correspondem aos requisitos da plataforma de modelação.

4. **Modelação dos Dados (Modelling)**

   O que torna a mineração de dados tão interessante é a existência de várias maneiras de resolver o mesmo problema. É nesta fase que temos de perceber os resultados da modelação, explorar inconsistências, aferir sobre a implementação dos mesmos na resolução do problema inicial, e se possível, comparar com outros sistemas de modelação. Se necessário, é preciso repensar o modelo e as técnicas de data mining que estão a ser utilizadas.

5. **Avaliação (Evaluation)**

   Depois da maioria do trabalho de mineração de dados estar efetuado, e antes de se passar para a fase de implementação, é necessário fazer uma avaliação aos resultados mediante o objetivo definido inicialmente. É a altura de fazer uma retrospeção acerca do que poderia ter sido melhorado ou se haveria outra alternativa, se houve alguma surpresa nos resultados obtidos, e se sim, se existe alguma forma de a prever.

6. **Implementação (Deployment)**

   Posteriormente à construção e avaliação do modelo(s), é crucial organizar e apresentar todo o conhecimento adquirido de uma forma a que o cliente/consumidor o entenda para poder assim utilizá-lo. Dependendo dos requisitos da aplicação, a fase de implementação pode ser tão simples como gerar um relatório ou tão complexa como implementar um processo de mineração de dados diferente. O objetivo é dar "vida" aos dados, para que sejam entendidos por todos.

2.3 Trabalho Relacionado

Esta tese tem um grande enfoque nas tecnologias para tratar grandes volume de dados em tempo real no contexto dos ITS, e nesta matéria já foram desenvolvidos vários trabalhos e estudos académicos. O congestionamento do tráfego é um problema sério e que assombra as infraestruturas de transporte de inúmeras estradas pelo mundo inteiro. Com perto de um bilião de veículos na estrada atualmente, e uma projeção da duplicação desse número para a próxima década [41], os atrasos excessivos causados pelo congestionamento não mostram sinais de diminuição num futuro próximo.

Por este motivo, e com a adoção generalizada de tecnologias de localização como o Global Positioning System (GPS), tanto nos automóveis como nos *smartphones*, o campo dos transportes inteligentes tem vindo a crescer e a despertar imenso interesse nos últimos anos. Neste domínio utilizam-se dados de localização em tempo real provenientes de várias fontes e conjugam-se os mesmos com dados estáticos de mapas ou de pontos de interesse, de forma a fornecer informação útil aos utilizadores.

A utilização crescente dos *smartphones* fez com que se criasse alguns sistemas de estimativa precisa do tráfego, utilizando para o efeito as capacidades dos mesmos. Um desses exemplos é o VTrack, um sistema em tempo real capaz de prever o tráfego baseado em sensores de GPS e Wi-Fi nos telemóveis, tendo em conta a energia utilizada no processo. Embora o GPS forneça estimativas de localização altamente precisas, tem várias limitações. Alguns telefones não têm GPS, e os que têm, por vezes a tecnologia não funciona, como ocorre em edifícios altos e túneis ou quando o telefone está dentro do bolso. Para além disso, o GPS na grande maioria dos telefones tem um gasto de bateria muito elevado. Nestes casos, o VTrack utiliza alternativas, menos sensíveis à energia, mas mais ruidosas como o Wi-Fi, para estimar a trajetória de um utilizador e o tempo de viagem ao longo da sua rota. No estudo levado a cabo pelo MIT (EUA) [43], os grandes desafios de utilizar os *smartphones* como provas para prever o tráfego estão essencialmente relacionados com dois fatores, o consumo de energia e as amostras imprecisas de localização. O VTrack utiliza o Hidden Markov Model (HMM) para modelar a trajetória de um veículo, associando cada amostra de posição ao ponto mais provável no mapa rodoviário. Este estudo chegou a duas conclusões fundamentais: a primeira é o facto dos tempos de viagem baseados apenas na localização Wi-Fi serem precisos o suficiente para o planeamento de rotas; enquanto que a segunda trata-se de quando o GPS está livre, ser conveniente a sua utilização periódica, alternando com o Wi-Fi, e obtendo assim melhores resultados no consumo de energia do *smartphone*.

A International Business Machines (IBM) é uma empresa americana que tem sido pioneira no estudo e desenvolvimento de estruturas para o avanço dos sistemas de transporte inteligentes. Um dos estudos realizados para esse desenvolvimento foi em Estocolmo, na Suécia, onde após largos anos a lutar contra o problema do elevado número de automóveis no centro da cidade, o governo sueco decidiu fazer um acordo com a IBM com o propósito de desenvolver um sistema capaz de taxar automaticamente todos os veículos registados

na Suécia se estes entrassem ou saíssem do centro da cidade, de Estocolmo, entre as 06h30 e as 18h30, excluindo fins de semana e feriados, e assim reduzir o congestionamento entre 10% e 15% [7].

A IBM utilizou etiquetas de Radio-Frequency IDentification (RFID), que usam ondas de rádio para identificar objetos, e sensores wireless, pequenos dispositivos que medem e detetam condições reais e convertem-nas em sinais que são enviados para os computadores. Desta forma, é possível a identificação das chapas de matrícula, muitas vezes difíceis de identificar devido ao mau tempo ou aos ângulos das fotografias capturadas. A IBM desenvolveu então um sistema de reconhecimento que recorre a algoritmos de processamento de imagem, para que através da comparação com placas traseiras e frontais se possa encontrar padrões, permitindo assim o seu reconhecimento. O sistema de DSMS escolhido para o processamento de *streams* para a aplicação foi o IBM Infosphere, que é o responsável pela pouca latência em processos como obtenção, limpeza, eliminação de ruídos e *matching* dos dados. Após identificada a matrícula esta é comparada com uma grande base de dados para posteriormente ser faturada. O objetivo e o processamento de dados desta aplicação é muito semelhante à da aplicação desenvolvida nesta tese. De facto, ambas partilham o processamento de dados em tempo real e a implementação de portagens dinâmicas. Tal como anteriormente referido, a aplicação protótipo desenvolvida, através do conhecimento adquirido, pode contribuir para a futura implementação de portagens dinâmicas, cenário cuja aplicação através da identificação automática de matrículas em tempo real, de certo modo também pretende atingir. Apenas é de realçar o uso de uma diferente ferramenta de processamento de *streams*, o IBM Infosphere em vez do Apache Storm.

Este teste teve a duração de seis meses e ficou estabelecido que no final haveria um referendo para a aprovação ou não do sistema desenvolvido. Este teste terminou em Julho de 2006, e acabou por exceder inclusivamente as suas expetativas, com a redução do congestionamento de tráfego na ordem dos 25%, os níveis de poluição a baixar de de 10% a 15% e os tempos de viagem mais rápidos e previsíveis. Desse modo, os suecos votaram positivamente no referendo previsto e o parlamento decidiu introduzir o sistema permanentemente. Esta foi a primeira vez na história em que residentes de uma cidade europeia decidiram aprovar uma lei que taxasse a utilização de estradas [20]. Após este projeto piloto, houve muitas cidades europeias que decidiram implementar o sistema de cobrança de taxas para diminuir o tráfego rodoviário, nomeadamente: Valleta (Malta), Londres e Durham (Inglaterra), Milão (Itália) e Gotemburgo (Suécia) [27].

Existe cada vez mais interesse em estimar o tempo de viagem dos motoristas, porém não existem muitos estudos que o façam em grandes zonas urbanas. O estudo "Large-Scale Estimation in Cyberphysical Systems Using Streaming Data: A Case Study With Arterial Traffic Estimation" [19] é um desses estudos, cuja implementação se baseia na muito recente ferramenta de processamento de *streams*, a framework *Spark Streaming*.

Para a previsão dos tempos de viagem foram utilizados dados de GPS, que representaram um enorme problema para o estudo devido à geração dos mesmo se realizar a uma

baixa frequência. Este fator, consequência dos constrangimentos de energia e de banda, fez com que os dados disponíveis não fossem muito confiáveis. No entanto, o intuito era testar a performance da ferramenta de processamento de *streams*, e por isso a fiabilidade dos dados não era o mais importante. O caso de estudo foi realizado em São Francisco (EUA) e teve em conta os dados provenientes de alguns táxis nessa cidade, fornecendo centenas de milhões de pontos de GPS. Foi então desenvolvido um algoritmo complexo EM (Expectation Maximization), que conseguiu provar que na cidade de São Francisco (EUA), este era capaz de processar dezenas de milhares de observações por minuto, com latência de apenas alguns segundos. É de realçar a importância que a ferramenta Spark Streaming teve para o sucesso do estudo, pois foi através da mesma que se pôde realizar uma computação distribuída por várias máquinas do algoritmo desenvolvido, permitindo assim uma latência baixa e uma elevada performance. Conclui-se então, que esta plataforma de processamento de *streams* é muito útil quando a necessidade é implementar algoritmos não triviais num grande *cluster*.

Para além das aplicações acima já desenvolvidas no âmbito dos transportes inteligentes, existem muitas outras fora deste contexto, mas que também utilizam o conceito de processamento de *streams* em tempo real.

Uma dessas aplicações é a monitorização da rede social Twitter. Como já referido anteriormente noutro Capítulo, o *Apache Storm*, tecnologia de processamento de *streams* utilizada neste trabalho, foi adquirida e desenvolvida pelo Twitter. O Storm atualmente é executado em centenas de servidores, distribuídos por todos os centros de dados da empresa. Várias topologias são executadas nesses *cluster* e algumas delas possuem centenas de nós, o que origina *Terabytes* de dados todos os dias, gerando biliões de *tuples* de output. Estas topologias são usadas para tarefas tão simples como filtrar, agregar conteúdo de várias fontes de *streams* do Twitter, como computar contagens de palavras de um certo tipo, quais os *tweets* mais publicados, as maiores tendências, entre outras. Contudo, podem também ser utilizadas para tarefas mais complexas como executar algoritmos de *machine learning*, como *clustering*. Para além do Twitter, empresas como a Groupon, Yahoo, Spotify, Alibaba, entre muitas outras, utilizam o Storm como ferramenta para analisar o imenso volume de dados que produzem diariamente.

Apesar de não ser um estudo de *stream processing*, o estudo que se segue teve uma grande importância para toda a comunidade de cientistas de dados, pois para além de ser um dos primeiros no contexto do *Big Data*, foi determinante para perceber algumas limitações da análise de uma grande quantidade de dados, principalmente através da Internet. Esse estudo, e um dos mais famosos é o *Google Flu Trends*, que como o nome indica tem como objetivo prever surtos de gripe, baseando-se para isso na pesquisa das pessoas no motor de busca *Google*. A cada dia, milhões de pessoas utilizam o *Google* para procurar informações que influenciam a sua vida, como por exemplo pesquisas de como tratar uma doença ou sintomas da mesma. Em 2008, pesquisadores da Google exploraram este potencial, afirmando que poderiam prever surtos de gripe com base na pesquisas dos utilizadores. O *Google Flu Trends (GFT)* utiliza dados agregados de consulta de pesquisa

para estimar a atividade da gripe nos Estados Unidos da América, comparando esses resultados com o dos serviços de saúde de todo o país. A ideia principal é que a maioria das pessoas quando estão ou pensam ter os sintomas de gripe, fazem pesquisas no *Google* acerca do assunto. Os estudos realizados, anteriormente à implementação da aplicação, revelaram que esses dados, se devidamente sincronizados com os Centros de Controlo e Prevenção de Doenças, poderiam produzir estimativas precisas do aparecimento de surto da gripe em tempo real, duas semanas mais cedo do que essa mesma instituição o faria [11].

Tentaram fazer o nunca antes imaginado, pesquisar as pesquisas, biliões delas, e analisaram-nas durante cinco anos na tentativa de descobrir algo. Foi então que fizeram uma descoberta, a análise de todos os dados mostrou que o número de pesquisas sobre gripe estava efetivamente correlacionado com as pessoas que realmente tinham a doença. Porém, em 2013 o resultado foi completamente diferente do esperado e a discrepância em relação aos dados do Centro de Controlo de Doenças dos EUA foi imensa. É essencial perceber o que aconteceu neste ano e que levou ao abandono completo do projeto por parte da Google, revelando ainda as inconsistências de algumas técnicas de análise de grandes dados. O que aconteceu foi que no ano de 2013 houve imensas notícias por parte dos meios de comunicação sobre a intensa temporada de gripe que se avizinhava. Este facto fez com que as pesquisas aumentassem sobre o assunto, fazendo com que os resultados fossem muito diferentes dos previstos.

É de salientar que a falha desta aplicação não apaga o valor do *Big Data*, apenas evidencia algumas das suas problemáticas. Este estudo foi extremamente importante para o despoletar de outros tantos nesta temática e atualmente existem inúmeros de aplicações semelhantes.

Capítulo 3

Fontes de Dados

Neste capítulo será abordado a fonte de dados disponível para a realização deste trabalho, bem como a descrição pormenorizada das suas características, desde a compreensão dos seus dados, qualidade, disponibilidade, até à identificação de problemas específicos, como por exemplo valores nulos, deteção de duplicados e incoerência de resultados. Além disso, existe um subcapítulo inteiramente dedicado à exploração dos dados, muito importante para traçar perfis/padrões de ocupação e velocidade das estradas. Destaca-se o uso do Tableau Desktop, um aplicativo avançado de descoberta e exploração de dados, para a análise dos dados ao longo deste Capítulo.

3.1 Data Understanding

3.1.1 Descrição dos Dados

Os dados disponíveis foram guardados numa base de dados MongoDB. O MongoDB é uma plataforma de banco de dados de alto desempenho, alta disponibilidade e simples escalabilidade, e que num só servidor normalmente tem várias bases de dados. Existem dois tipos de bases de dados, no que diz respeito ao seu modelo relacional, bases de dados relacionais e não relacionais. O Mongo pertence ao grupo das bases de dados NoSQL (Not Only SQL), bases de dados não relacionais. A propriedade principal deste tipo de bases de dados é que não utilizam a linguagem SQL, não seguindo o modelo relacional, típico da mesma. Este tipo de bases de dados não estão assim obrigadas a utilizar uma rede de tabelas interligadas por chaves, sendo por isso muito mais flexíveis, podendo adicionar dados mais facilmente sem ser necessário reestruturar todas as tabelas. Enquanto o modelo relacional apresenta a informação sobre a forma de uma lista, o NoSQL armazena as informações a serem apresentadas em conjunto, o que facilita a sua consulta às mesmas.

Este sistema de arquivos funciona assente em dois conceitos essenciais, o de coleção e o de documento. Uma coleção existe dentro de uma única base de dados e é constituída por um conjunto de documentos, analogamente ao que uma tabela é nas tradicionais Relational Database Management System (RDBMS).

Um documento é um conjunto de pares chave-valor. É de destacar que cada documento tem obrigatoriamente um campo com o nome *_id* cujo valor é um número hexadecimal de 12 bytes que pode ser gerado aquando da inserção pelo utilizador ou, caso contrário, é automaticamente gerado pelo Mongo e que assegura a unicidade de cada documento dentro de uma coleção. Os documentos têm um esquema dinâmico, o que significa que documentos na mesma coleção não têm de ter necessariamente o mesmo conjunto de campos ou estrutura. Os documentos são representados em formato JSON. Esta notação de objetos JSON (JavaScript Notation Objects) juntamente com XML, são os formatos principais para o intercâmbio de dados nos browsers modernos. O JSON suporta topo o tipo de dados: números, strings, datas, *doubles*, valores booleanos, matrizes, *hashmaps*, entre outros. Na Figura 3.1 apresenta-se a título de exemplo geral a notação JSON para uma melhor compreensão de que forma se encontram guardados os dados dos sensores no MongoDB.


```
{
    "_id" : ObjectId("598342abed57b82a28f7dfd8"),
    "properties" : {
        "stevci_gap" : "15,4",
        "stevci_statOpis" : "Normalen promet",
        "stevci_hit" : "114",
        "stevci_stev" : "240",
        "stevci_pasOpis" : "(v)",
        "stevci_smerOpis" : "Celje zahod - Celje",
        "stevci_stat" : "1"
    },
    "Id" : "0648-21",
    "ReadingPoint_id" : "5919e2c9452fbb232810f9ab",
    "ModifiedTime" : ISODate("2017-04-03T14:38:31.000+0000")
}
```


Figura 3.1: Exemplo Notação JSON

No que diz respeito ao *Big Data*, esta base de dados é frequentemente utilizada em detrimento de outras existentes. Existem vários estudos, [31], [48], [49], que comparam o MongoDB, uma base de dados NoSQL, com outras bases de dados (SQL e NoSQL), e que concluíram que esta apresenta vantagens em termos de performance, segurança, integridade de dados e interoperabilidade em relação às restantes. Através desta conclusão pode-se afirmar que este tipo de base dados é o indicado quando queremos fazer consultas e inserções a uma grande velocidade numa grande quantidade de dados, mantendo a segurança e integridade dos mesmos, requisito obrigatório na implementação pretendida.

Os dados são provenientes de sensores localizados nas estradas da Eslovénia. Este país tem uma área de 20.273 km², aproximadamente 4,5 vezes menor que Portugal, possui cerca de 2.065.895 habitantes e cerca de 1.412.315 de veículos motorizados registados, dados de 2014 provenientes do Gabinete Oficial de Estatística Esloveno [37]. Os dados

representam medições de 349 sensores diferentes desde Janeiro de 2016 até Maio de 2017, 17 meses de dados, perfazendo no total cerca de 33 *Gygabytes* de informação.

Quanto à sua localização e distribuição, podemos afirmar que, apesar de cobrirem a maior parte do território do país, se concentram em maior número nas áreas circundantes e mesmo nas proximidades da capital, Liubliana, e da segunda maior cidade da Eslovénia, Maribor, como pode ser observado na Figura 3.2.

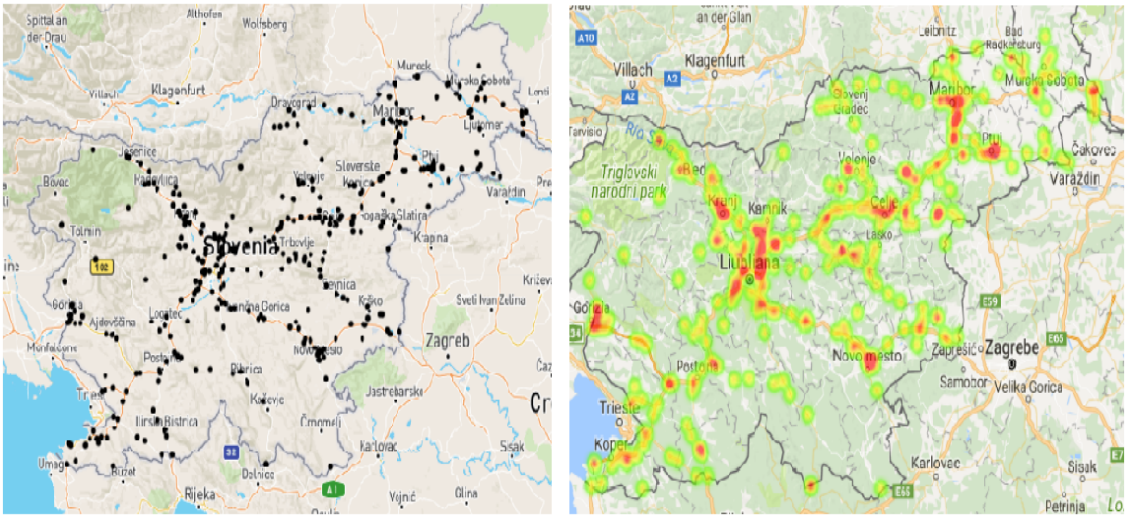


Figura 3.2: Localização e distribuição geográfica - Sensores Eslovénia

A ilustração deste mapa só foi possível devido a informações sobre a localização dos sensores (latitude e longitude) presente na base de dados. Efetivamente para uma melhor perceção da riqueza dos nossos dados é primordial saber o seu significado e que unidades representam, ainda para mais neste caso específico em que todo o *dataset* se encontra em esloveno. Por conseguinte, passemos a descrição do *schema* e dos meta-dados existentes dos mesmos.

Cada documento de MongoDB contém todos os dados que correspondem à medição de certos índices, num determinado intervalo de tempo, normalmente 5 ou 10 minutos para todos os sensores. Estes sensores, identificados por um número único correspondente ao campo *Id*, contêm os valores divididos por faixas e por direção, ou seja, existem sensores com apenas duas faixas e outras com quatro, sendo que estas contêm uma direção específica no trânsito, como por exemplo de Norte para Sul ou vice-versa. É de salientar que no máximo existem três faixas de trânsito por direção em toda a base de dados.

Vejamos todos os campos presentes na nossa base de dados na forma de Tabela 3.1. Como podemos constatar possuímos um conjunto vasto de meta-dados que fornecem inúmeras informações, tais como a caraterização das estradas, nomeadamente nome, local, localização exata dos sensores (latitude e longitude), e outras como a velocidade máxima permitida, a data e hora a que foi efetuado o registo, e também todos os índices medidos pelo sensores como velocidade, o intervalo de tempo entre veículos, a ocupação, todos eles por faixa e direção como já referido.

A qualidade de todos estes dados irá ser discutida no próximo subcapítulo.

Tabela 3.1: Descrição dos meta-dados

Meta-dado	Descrição	Tipo	Unidade
ModifiedTime	Data e hora da última modificação	String	——
updated	Número total de segundos desde 1 Jan de 1970	Inteiro	——
copyright	Direito Autoral - Centro de Informações de Tráfego das Estradas Públicas Esloveno	String	——
Title	Nome completo da estrada	String	——
Id	Id único do sensor, e.g.: 0178	String	——
x_wgs	Longitude de onde está localizado o sensor	Double	——
y_wgs	Latitude de onde está localizado o sensor	Double	——
stevci_cestaOpis	Nome da estrada	String	——
stevci_vmax	Limite de velocidade da secção de estrada onde o sensor está inserido	Inteiro	km/h
stevci_odsek	Número da secção onde a estrada está inserida	Inteiro	——
stevci_regija	Nome da Região na Eslovénia	String	——
stevci_stacionaza	Distância desde o início da estrada até à posição do sensor	Double	Km
id	Id do sensor concatenado com a direção da estrada e faixa, e.g.: 0178-12 sendo que 1 é a direção e o 2 representa a faixa	String	——
stevci_gap	Intervalo de tempo entre veículos	String	segundos
stevci_hit	Velocidade média dos veículos	String	Km/h
stevci_pasOpis	Descrição da faixa do sensor, e.g.: (v)- faixa mais à direita; (p)- faixa de ultrapassagem	String	——
stevci_smerOpis	Descrição do local na Eslovénia onde a estrada começa e acaba	String	——
stevci_stat	Número de 1 a 6, que descreve o estado do trânsito, sendo que 1 corresponde a pouco trânsito e 6 a trânsito intenso	String	——
stevci_statOpis	Descrição do estado do trânsito, e.g.: gost promet- trânsito intenso	String	——
stevci_stev	Ocupação da estrada durante o intervalo de tempo, número de viaturas nesse intervalo	String	——

3.1.2 Qualidade e Disponibilidade dos Dados

Uma das principais fases no que toca ao Data Understanding é a qualidade e disponibilidade dos dados. Se a disponibilidade e qualidade dos dados que temos à nossa disposição for muito deficiente, isto é, se tivermos poucos dados ou se estes apresentarem características anormais, não poderemos produzir resultados confiáveis e nem validar por vezes teorias ou conceitos, que no caso de termos um ótimo *dataset* se confirmariam. Os dados que estamos a utilizar são adquiridos de sensores que estão colocados em várias estradas, sujeitos a intempéries, atos de vandalismo, falta de manutenção e por estes motivos altamente vulneráveis a erros. Efetivamente, os dados do mundo real são imperfeitos, podemos ter dados incompletos como falta de atributos ou dos seus valores, *noisy data*, isto é, que contêm erros ou valores atípicos, ou até mesmo dados inconsistentes e que possuem discrepâncias em códigos, números ou nomes. Por todas estas razões temos de ter em conta estes fatores no cálculo dos índices de disponibilidade e qualidade dos dados, de forma a que as conclusões deste estudo possam ter em conta os mesmos.

Antes de passarmos à análise da qualidade dos dados, e até porque esta está diretamente relacionada com a disponibilidade dos mesmos, é de realçar que em relação a esta última característica em 2016 contamos com aproximadamente 23,5 milhões de registos enquanto que em 2017 registaram-se 5,2 milhões ao longo de apenas os primeiros cinco

meses do ano, totalizando perto de 29 milhões de registos em toda a base de dados. Como podemos perceber temos um enorme conjunto de dados e que prova a inserção no contexto de *Big Data* deste projeto, porém esta disponibilidade não se encontra igualmente distribuída por todos os meses, pelo que se destaca a diferença entre os mesmos na Figura 3.3.

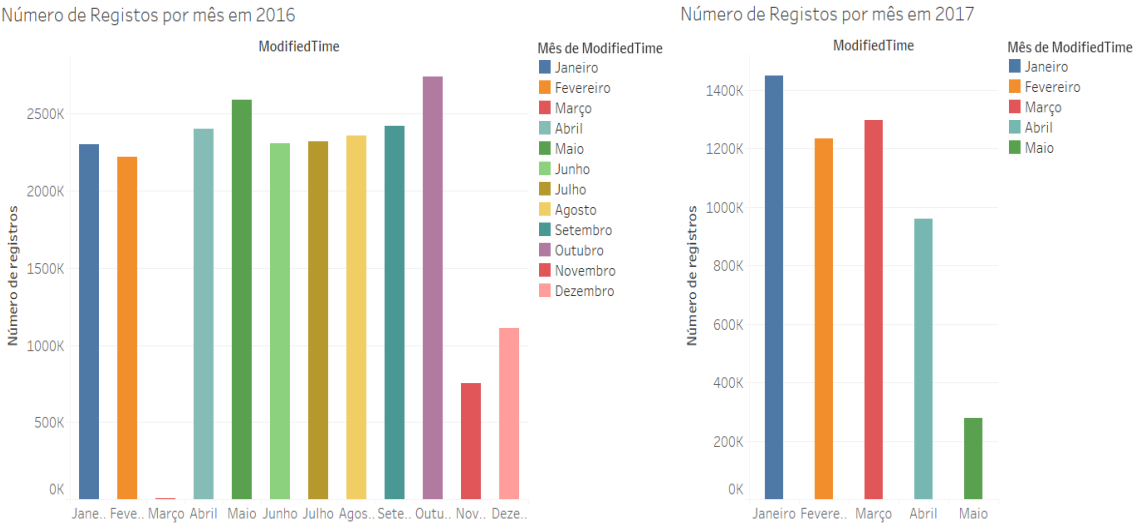


Figura 3.3: N.º de Registos por mês desde Janeiro 2016 até Maio 2017

Como podemos atentar, o mês de Março é definitivamente o mês com menos dados, distanciando-se imenso de todos os outros no que diz respeito à sua disponibilidade. É essencial perceber que estes valores por vezes diferem, só pelo simples facto do *input rate* dos registos se realizar em dois tempos diferentes, 5 e 10 minutos, como já assinalado. Consideremos então os dois casos para perceber qual a diferença no número de registo que esperaríamos obter. No primeiro caso, e considerando que um mês tem 30 dias, por sensor, deveríamos obter 8.640 registos e no segundo caso 4.320 registos por mês, valores de referência para um sensor caso não haja falhas nas medições.

Tomando o mês de Janeiro de 2016 como exemplo, e atendendo à circunstância de que a receção de informação do mesmo é de 5 em 5 minutos, por mês o número esperado de registos seria de 2.393.280 milhões, tendo este 277 sensores a debitar dados. Tal não acontece, dado que o valor de registos é de 2.303.659 milhões, porém podemos concluir que com uma taxa de 96,2% de disponibilidade, a qualidade deste mês, no que diz respeito à quantidade de informação, é bastante satisfatória.

Atendendo ao *input rate* e ao número de sensores presentes em cada mês, pode ser vista a disponibilidade dos dados em percentagem na Tabela 3.2. Podemos verificar que abaixo dos 80% de disponibilidade temos apenas 4 meses. O mês de Março e Novembro de 2016 apresentam esta percentagem, pois apenas têm valores para 1 dia e para 9 dias, respetivamente. Este acontecimento pode ter inúmeras explicações, nomeadamente o mau funcionamento dos sensores, reparação ou manutenção dos mesmos, falta de energia ou mesmo má comunicação entre o envio e o recebimento dos dados por parte da base de

dados. O mês de Maio de 2017 apresenta o mesmo problema acima explicitado, contando apenas com registos para alguns dias.

Tabela 3.2: Percentagem por mês da disponibilidade dos dados

Mês	**Disponibilidade por mês [%]**	**Taxa de input rate [min]**
Janeiro 2016	96,2	5
Fevereiro 2016	91,4	5
Março 2016	0,25	5
Abril 2016	98,6	5
Maio 2016	99,2	5
Junho 2016	88,4	5
Julho 2016	88,8	5
Agosto 2016	90,4	5
Setembro 2016	92,7	5
Outubro 2016	96,4	5
Novembro 2016	26,4	5
Dezembro 2016	86,3	10
Janeiro 2017	97,2	10
Fevereiro 2017	82,7	10
Março 2017	86,4	10
Abril 2017	63,8	10
Maio 2017	18,4	10

No caso concreto do mês de Abril de 2017, a questão é diferente, pois apesar de não conter os primeiros dois dias do mês, apresenta registos em todos os outros, mas estes apresentam anomalias no que diz respeito ao número de registos. Ora vejamos, a taxa de input deste mês são de 10 em 10 minutos, o que significa que por sensor, por dia, deveríamos obter 144 registos e isso não se verificou na análise realizada. Podemos perceber pela Figura 3.4, e ressalvando que esta análise é realizada apenas a um sensor, que no mês de Abril nos dias 5, 9, 11, 15 e 23 o número de registos difere entre 13 e 14. Isto deve-se ao facto de que nesses dias as únicas horas em que foram realizadas medições foram às 21, 22 e 23 horas, como podemos confirmar na imagem subjacente no dia 5 e 15 de Abril. Esta constatação repete-se igualmente nos restantes dias mencionados onde existe esta deficiência de número de registos.

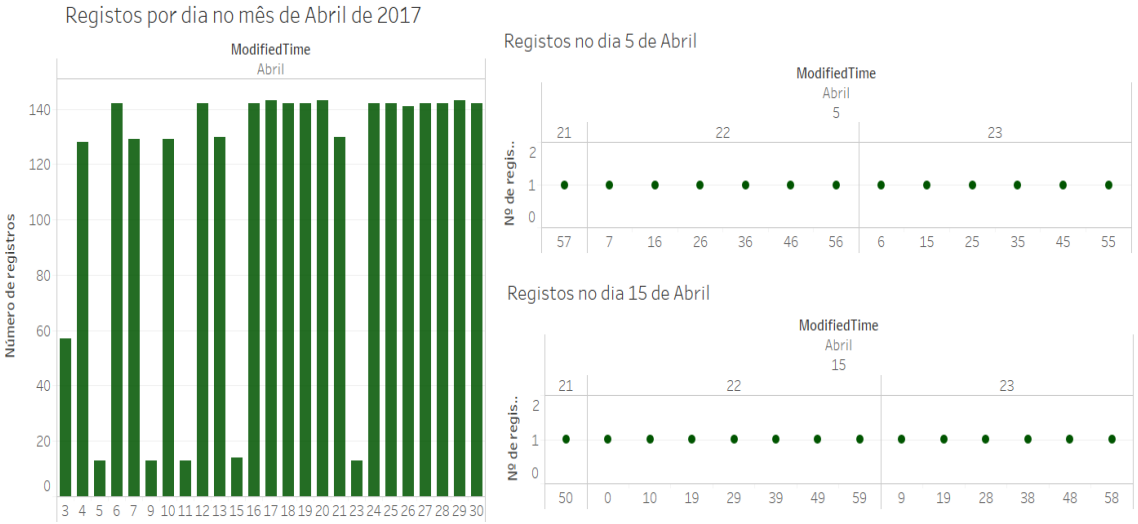


Figura 3.4: N.º registos de um único sensor no mês de Abril de 2017

Para além desta falta de registos foram identificadas outras situações onde foram detetadas algumas anomalias na medições dos dados. Algumas dessas situações serão explicitadas de seguida:

1. Diferença de tempo entre veículos discordante com a ocupação da via;

   Aquando da comparação do índice de diferença de tempo entre veículos (*stevci_gap*), medido em segundos, com a ocupação da estrada (*stevci_stev*) encontrou-se situações onde o tempo registado não poderia corresponder à verdade em comparação com o número de veículos que circulava num determinado período de tempo. Um desses exemplos pode ser visto na Figura 3.5, e onde se destaca a anotação presente na mesma. No dia 21 de Março de 2017, à 1 hora e 16 minutos, a diferença de tempo entre veículos registada foi de 999 segundos, o que não pode ser possível tendo em conta que o número de veículos foi de 24, no mesmo período de 10 minutos.

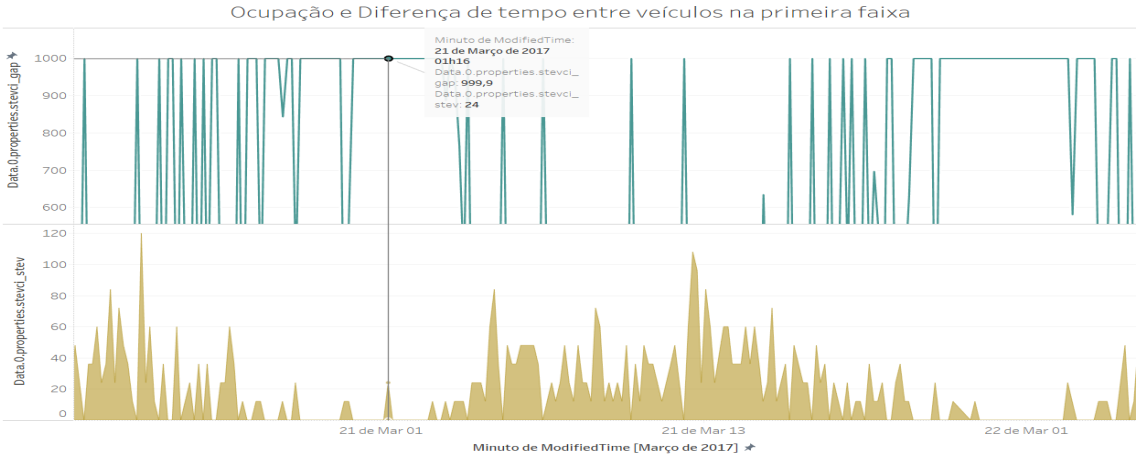


Figura 3.5: Comparação entre a diferença de tempo entre veículos e a ocupação da via

2. Divergência entre valores de estado do trânsito em relação à ocupação da via;

   O meta-dado *stevci_stat* representa, através de um número inteiro de 1 a 6, o estado do trânsito na via, sendo que este é maior quanto maior for esse número. Todavia, verificaram-se ocorrências em que a regra anterior não se aplica. Na Figura 3.6, abaixo representada, podemos identificar episódios que quebram a regra, como por exemplo, à 1 hora da manhã do dia 27 de Abril, onde o estado do trânsito é de 2 e a ocupação de 12 veículos, e na hora seguinte o estado é decrementado e contrariamente ao esperado a ocupação aumenta para 132 veículos.

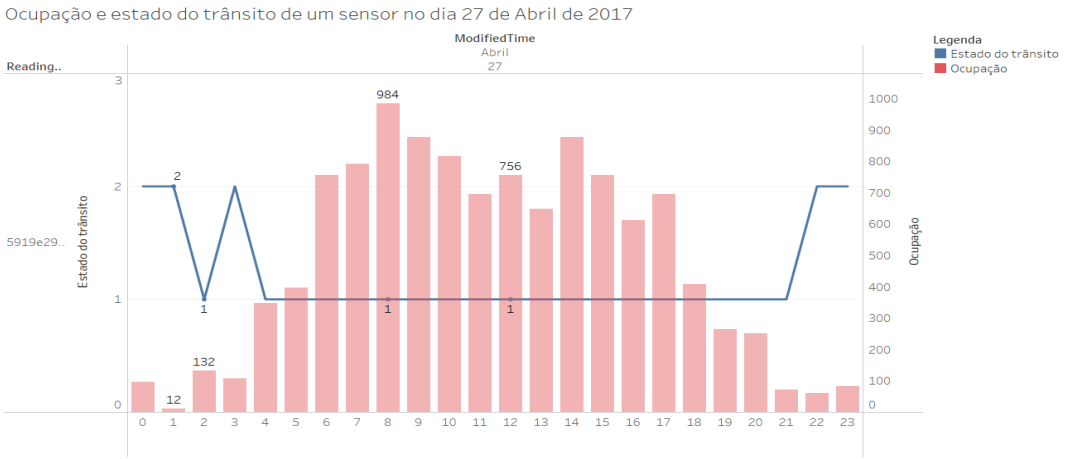


Figura 3.6: Relação entre o estado do tráfego e a ocupação da via

3. Velocidades médias não concordantes com o número de carros a fluir;

   Foram distinguidos alguns eventos onde esta discordância se verificou, e onde o caso exposto na Figura 3.7 se inclui. Nas barras a azul podemos observar a velocidade média, e nos traços horizontais a preto a ocupação para registos de 10 em 10 minutos a partir das 23 horas do dia 19 de Janeiro de 2017. Conseguimos identificar então duas situações que levaram ambas a interrogações pertinentes: como é possível que num período de 10 minutos exista uma velocidade média de 255 km/h quando o número de veículos é de 12, e também como é que a velocidade média pode ser de 2 km/h quando o fluxo de automóveis é de aproximadamente 50. De facto, estas velocidades não correspondem de todo a esta ocupação no período de 10 minutos estudado. Este erro tem a ver com o facto da velocidade não ser medida, mas sim calculada em função do número de carros medido, ou seja, quanto maior for a ocupação menor a velocidade, e no limite se não passar carro nenhum a velocidade tende para infinito.

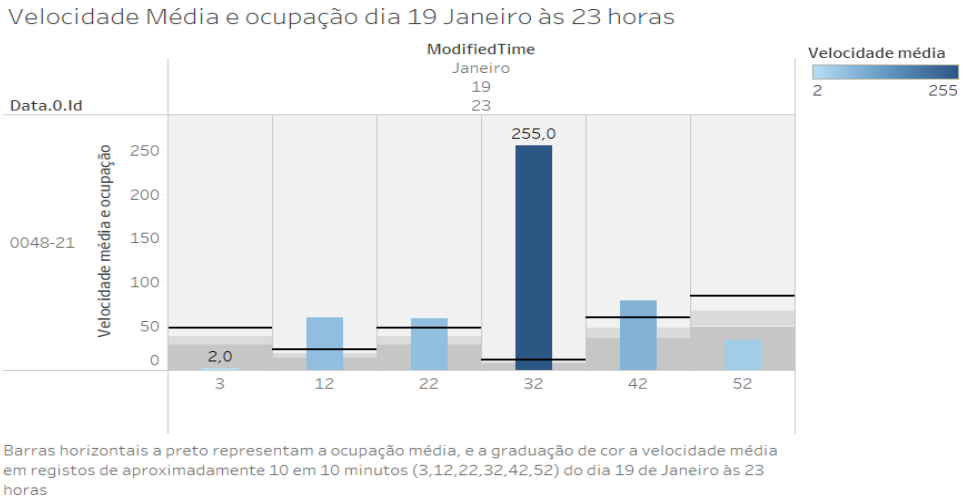


Figura 3.7: Contraste entre velocidade média e ocupação da via

4. Descrição do estado do trânsito ambígua quando equiparada à ocupação da via;

   Como identificado na Tabela 3.1, existem dois parâmetros capazes de descrever o estado do trânsito, um em forma de numeração e outro, em questão, de uma forma descritiva (*stevci_statOpis*). Para se compreender melhor vejamos a tradução numa pequena Tabela 3.3, dos vários estados de trânsito possíveis. De facto, só pela observação da mesma se verifica que existem campos cuja descrição é muito semelhante, e que por isso, não chega para tirar ilações sobre a verdadeira descrição do estado do trânsito. Para além disso, ainda se torna mais evidente a ambiguidade quando temos ocupações semelhantes e estados de tráfego diferentes, como pode ser visto na Figura 3.8.

   Podemos entender que não se trata efetivamente de um erro nos dados, porém é desconhecido inteiramente como é realizada esta classificação, e por essa razão, não podemos utilizar estas descrições para tirar quaisquer conclusões.

Tabela 3.3: Tradução do estado do tráfego rodoviário de Esloveno para Português

Stevci_stat (Esloveno)	Estado do trânsito (Português)
gost promet	tráfego intenso
povečan promet	aumento de tráfego
zgoščen promet	tráfego condensado
gost promet z zastoji	congestionamento do tráfego pesado
normalen promet	tráfego normal

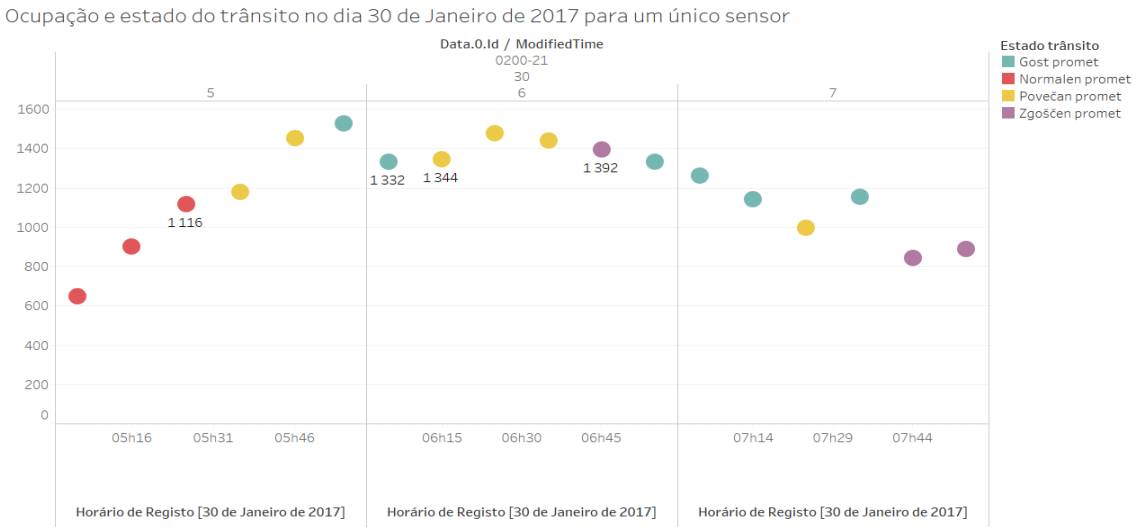


Figura 3.8: Comparação entre a descrição do estado do tráfego e a ocupação da via

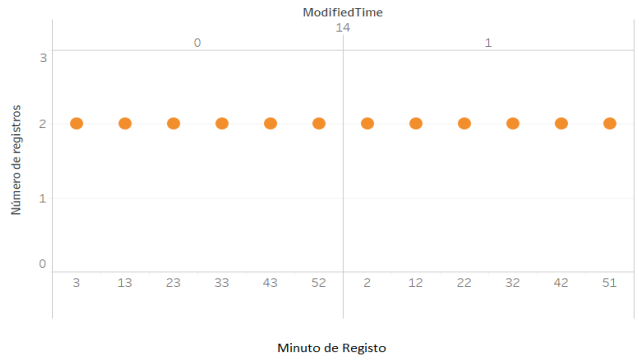
3.2 Data Selection

Tendo em consideração todo o conhecimento alcançado acerca da qualidade e disponibilidade dos dados no capítulo anterior, foi necessário realizar uma limpeza e seleção aos mesmos.

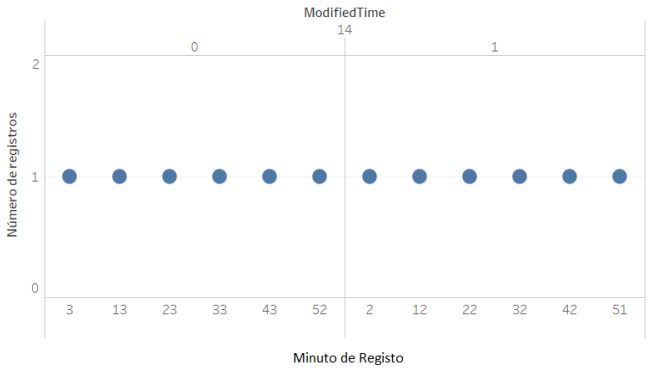
Ao início, os dados no seu estado mais "cru", apresentavam muitos registos repetidos e campos que não eram relevantes para análise ou cujas medições não eram as mais corretas. Desta forma, e posteriormente a análise dos dados, foram eliminados alguns campos e reorganizados os registos, cuja descrição detalhada de todo o processo se prossegue abaixo.

3.2.1 Limpeza de Dados e Seleção Final

Um dos principais procedimentos que foi realizado na limpeza dos dados foi a eliminação de registos duplicados, dado que a sua prevalência em nada beneficia à análise aos mesmos. Através do campo *Modified_Time*, que indica a data e hora na qual o registo foi efetuado, foi possível esta operação. Tomando o mês de Março de 2017 como exemplo, que tal como todos os outros meses continha registos repetidos, podemos destacar o facto do número de registos antes da limpeza dos duplicados ser de 1.432.416 milhões de registos e posteriormente se fixar em 1.296.392, aproximadamente 140 mil registos eliminados. Na Figura 3.15 podemos observar um exemplo desta duplicação de dados no dia 14, que como se pode deduzir, todos os registos antes da limpeza durante a meia noite e a uma da manhã estavam duplicados. Como este caso existem vários neste mês e noutros meses, e por essa razão, este foi o primeiro passo tomado antes da escolha final dos dados a utilizar.



(a) Registos de Março 2017 antes da remoção de duplicados



(b) Registos de Março 2017 após a remoção de duplicados

Figura 3.9: Comparação entre os registos antes e após a limpeza dos dados

Após esta limpeza aos dados seguiu-se a fase de seleção dos dados a reter, esta decisão teve em consideração sobretudo sobre que valores a ferramenta de processamento de *streams* iria utilizar para a análise. Em relação a este tema foram efetuadas três operações principais.

A primeira foi retirar de cada mês toda a informação relativa à caracterização exclusiva dos sensores, ou seja, campos como o *Id*, a localização, descrição, o título, a localidade, entre outros, foram retirados para uma coleção distinta e armazenados em vários documentos, tendo como critério de unicidade o *Id* do sensor. Esta operação foi realizada por uma questão de organização da base de dados e por facilitar a visualização geográfica da localização de todos os sensores, bastante útil na fase de visualização dos dados.

A segunda operação foi a escolha dos campos a absorver e de que forma os mesmos ficariam dispostos nos documentos das coleções de cada mês. Enquanto que, numa primeira fase cada documento continha os dados de todos os sensores para um determinado *Modified_Time* dentro de um *array*, a escolha nesta fase recaiu por dividir cada sensor por um documento com os campos pretendidos. A triagem dos dados para cada sensor, dentro dos vários meta-dados à disposição foi realizada tendo em conta a qualidade e a importância dos mesmos para a aplicação a ser desenvolvida. Desta forma, por documento os dados possuem 5 campos: o *Modified_time*, com a data e hora do registo; o *_id*, chave única do MongoDB; o *Id* do sensor; o *ReadingPoint_id*, que é o campo que faz a correspondência

entre as coleções de cada mês com a coleção referida na primeira operação; e por fim, o mais importante o campo *Data*, *array* que contém as medições do sensor. Este *array* tem tantas posições quantas faixas tem o sensor, normalmente duas ou quatro, havendo excecionalmente sensores com três ou cinco faixas. Dentro de cada posição do *array Data* temos o campo *Id*, mas que desta vez tem concatenado juntamente com o mesmo a direção e faixa do sensor (e.g.: 0180-12), e o objeto *properties* que contém os campos *stevci_gap* (diferença de tempo entre a passagem de veículos), *stevci_statOpis* (descrição do estado do trânsito), *stevci_hit* (velocidade), *stevci_stev* (ocupação da via), *stevci_pasOpis* (descrição da faixa do sensor), *stevci_smerOpis* (localidade de começo e término da estrada onde se encontra o sensor) e *stevci_stat* (descrição do estado do trânsito de forma numérica). Para além disso, os registos que estavam anteriormente desordenados, isto é, não estavam dentro da coleção por ordem ascendente de data e hora, foram colocados dessa forma.

Por último, mas não menos importante, foi realizada operação de conversão de tipos de dados. Na verdade termos a velocidade de carros ou a ocupação da via representada pelo tipo string, ou mesmo a data e hora, dificulta imenso o trabalho de ordenação e análise dos dados. Por esse motivo, a ocupação e velocidade foram convertidas para um número inteiro, e a data e hora (*Modified_Time*) foi convertido para o tipo *Date*, todos eles tipos de dados suportados obviamente pela base de dados MongoDB. Após esta limpeza e ordenação de dados de todos os meses, como pode ser observado na Figura 3.10, considerou-se terminado este trabalho que originou a configuração final dos dados, que será posteriormente utilizada para a ferramenta de processamento de *streams* em tempo real, o Apache Storm.


```
{
    "_id" : ObjectId("5911aee968af5f213c9ecbe5"),
    "Data" : [
        {
            "properties" : {
                "stevci_gap" : 3.6,
                "stevci_statOpis" : "Zgoščen promet",
                "stevci_hit" : NumberInt(98),
                "stevci_stev" : NumberInt(888),
                "stevci_pasOpis" : "(v)",
                "stevci_smerOpis" : "Barjanska - Peruzzijeva",
                "stevci_stat" : "3"
            },
            "Id" : "0178-21",
            "Icon" : "3"
        },
        {
            "properties" : {
                "stevci_gap" : 7.1,
                "stevci_statOpis" : "Normalen promet",
                "stevci_hit" : NumberInt(131),
                "stevci_stev" : NumberInt(492),
                "stevci_pasOpis" : "(p)",
                "stevci_smerOpis" : "Barjanska - Peruzzijeva",
                "stevci_stat" : "1"
            },
            "Id" : "0178-22",
            "Icon" : "1"
        }
    ],
    "Id" : NumberInt(178),
    "ReadingPoint_id" : "58f7f5a1754aa90d30a3e1f8",
    "ModifiedTime" : ISODate("2017-09-13T16:15:35.737+0000")
}
```


Figura 3.10: Formato dos dados no MongoDB após processo de limpeza e seleção

3.3 Exploração dos Dados

Esta secção dedica-se inteiramente à exploração dos dados, desde a descoberta de padrões, como os de sazonalidade, irregulares ou regulares de ocupação e velocidade, e se possível com esta exploração, o traçar de perfis de utilização das estradas na Eslovénia.

Comecemos pela ocupação média das estradas na Eslovénia ao longo do ano. A Eslovénia devido à sua localização geográfica, faz fronteira com 4 países: Áustria, Croácia, Hungria e Itália. Existem várias estradas que fazem ligação aos mesmos, onde se localizam os sensores, o que pode significar um aumento do número de veículos nos meses de maior calor, pelo simples facto das pessoas que se deslocam para os países do sul da Europa terem de passar pelo país em questão. Por outro lado, como o turismo recai mais sobre as capitais, Luibliana, ou as maiores cidades, Maribor, e como os sensores se encontram em maior número em ambas ou nas suas proximidades, o valor da ocupação também pode aumentar.

Os dados que dispomos vão de encontro ao que foi dito e para o comprovar vejamos a ocupação média por mês das duas faixas numa só direção, de um subconjunto de sensores localizados em estradas que têm comunicação com os países enumerados acima, durante todo o ano de 2016, na Figura 3.11. Podemos observar que o padrão de ocupação nas duas faixas, representadas por cores diferentes, é semelhante e que o mês onde há maior ocupação é o Agosto, o pico do Verão. Contudo, destaca-se também o aumento significativo de Junho para Julho e o decréscimo com a chegada do Outono, de Setembro para Outubro. É de salientar também a Figura 3.12 que apresenta a localização dos 18 sensores utilizados para esta análise.

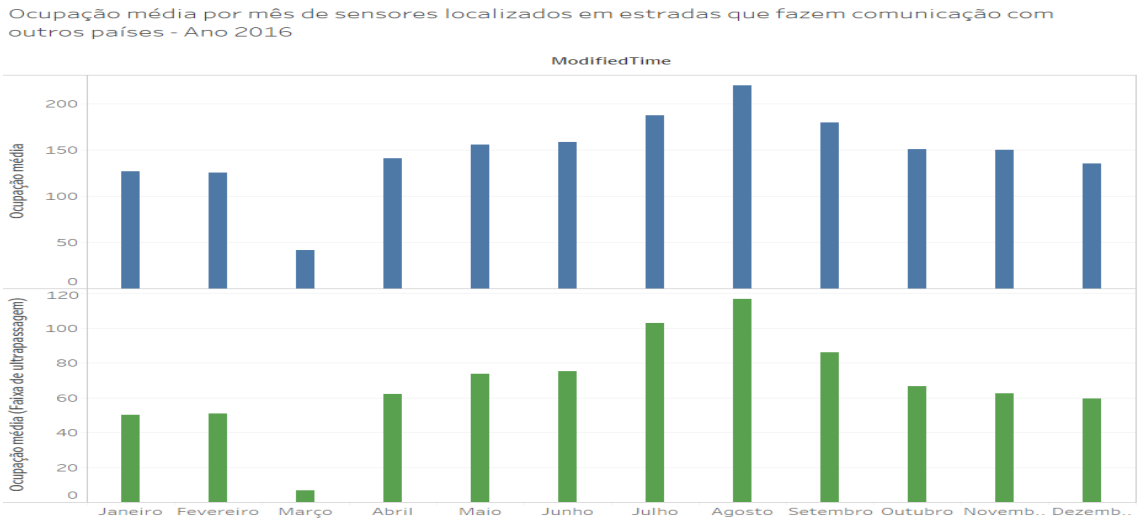


Figura 3.11: Ocupação média por mês de sensores localizados em estradas com comunicação com outros países - Ano 2016

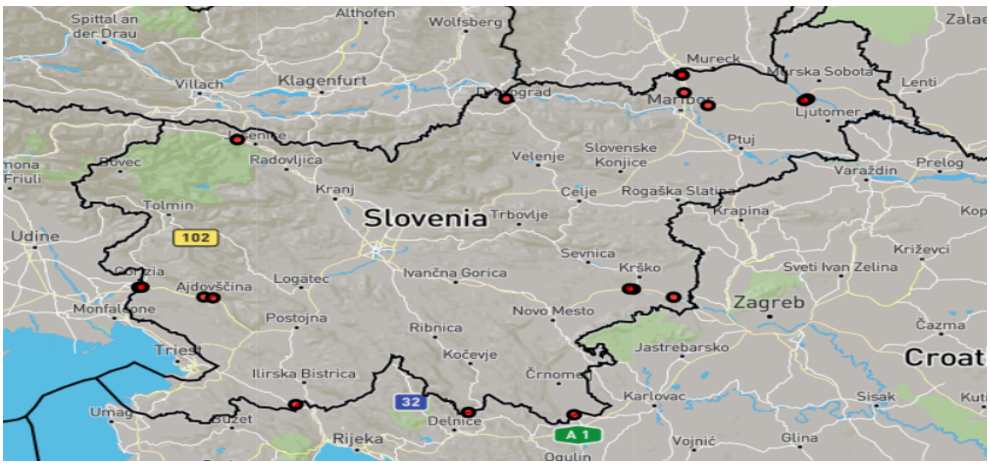


Figura 3.12: Localização dos sensores utilizados para analisar o padrão anual de ocupação na Eslovénia - Ano 2016

Relativamente à ocupação média durante as várias horas do dia registou-se um padrão algo incomum quando comparado com a maioria de outros países europeus, fundamentalmente durante o período da tarde. De sublinhar que os dados para esta comparação foram retirados da plataforma TomTom City [44], da empresa TomTom, uma marca de referência em produtos de navegação, trânsito e cartografia. Em Portugal, por exemplo, as horas de maior afluência de viaturas na capital, Lisboa, dados do portal TomTom City de 2016, são no período da manhã das 8 às 9 horas e no período da tarde das 18 às 19 horas.

Na análise realizada a este padrão na Eslovénia chegou-se à conclusão que o período de maior tráfego na maioria dos meses de Janeiro a Maio de 2017, se regista da parte da manhã das 6 às 7 horas e no período da tarde das 14 às 15 horas, como ilustrado na Figura 3.13. Comparando estes valores com os do TomTom City para Liubliana em 2016, identificamos que no período da manhã os maiores volumes de congestionamento se registam duas horas depois, das 8 às 9 horas, e no período da tarde também duas horas mais tarde, das 16 às 17 horas.

Este acontecimento pode dever-se ao facto de que a análise do portal TomTom é realizada em 25 pontos exclusivamente no interior da capital, enquanto que os sensores da aplicação analisados são em média 300 pontos em cada mês e estão dispersos por todo o tipo de estradas situadas um pouco por toda a Eslovénia, o que pode alterar ligeiramente este padrão.

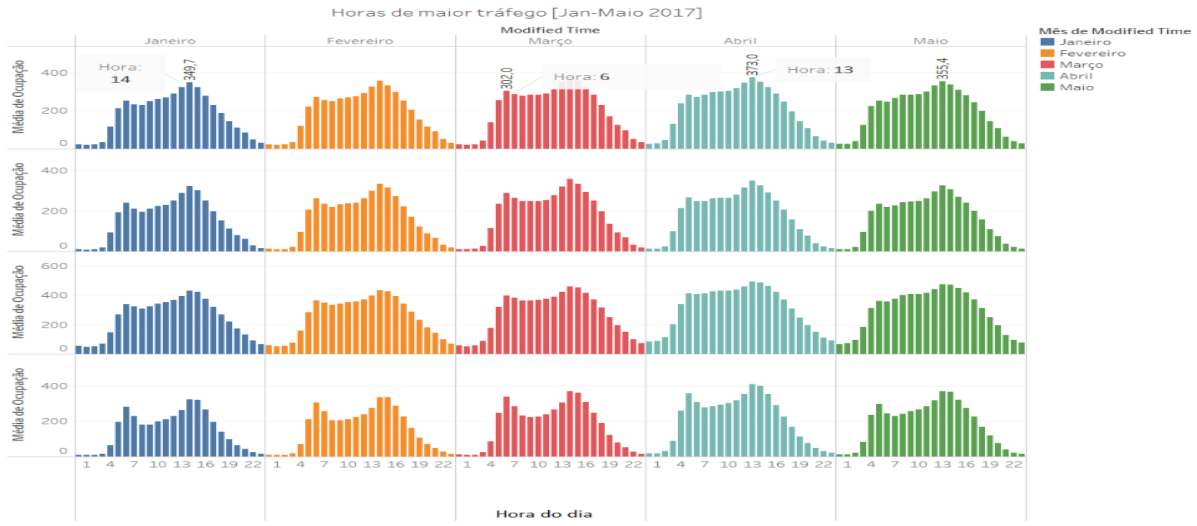


Figura 3.13: Ocupação média por hora de todos os sensores - Ano 2017

Outro dos padrões de ocupação encontrados está relacionado com a diminuição do tráfego durante o fim de semana, como seria de esperar. Durante os dias da semana a ocupação é praticamente igual em todos eles, mas durante Sábado e Domingo regista-se um pequeno decréscimo. Podemos observar um exemplo dessa diminuição na semana de 14 a 22 de Fevereiro de 2017 na Figura 3.14, ilustrada abaixo. Enquanto que existe sempre uma ligeira diminuição do tráfego durante o fim de semana pode também ser observado o aumento do mesmo à sexta-feira, que não se verifica apenas nesta semana mas também em diversas outras durante todos os meses analisados. Este episódio também apresenta uma explicação lógica. A sexta-feira, principalmente durante o período da tarde, é o dia em que as pessoas optam por realizar atividades lúdicas, como a ida a restaurantes, cinema, atividades ao ar livre, entre outras, por no dia seguinte não ser dia de trabalho. Além do mais é também neste dia que algumas pessoas se dirigem para fora das cidades, por exemplo para um fim de semana perto da praia ou de visita a localidades onde residem familiares, etc.

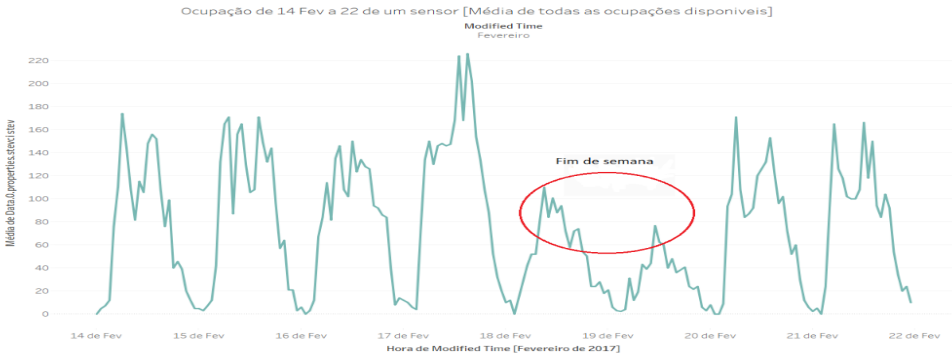
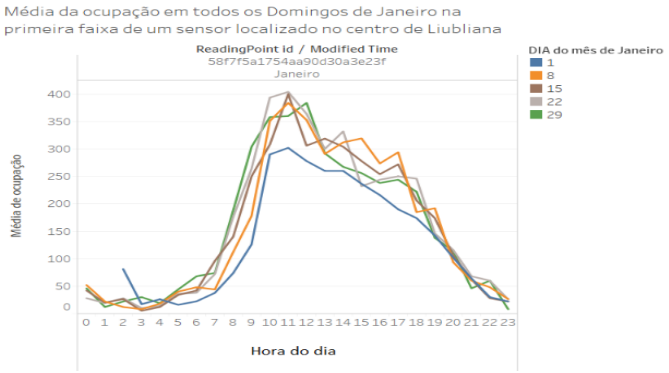


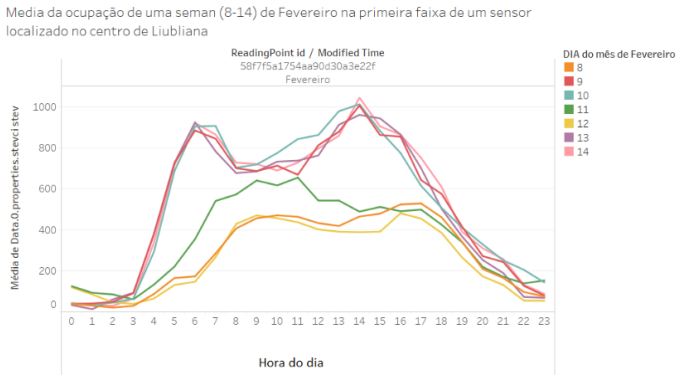
Figura 3.14: Ocupação média por semana (14 - 22 de Fevereiro) - Ano 2016

Todavia, existem situações em que estes padrões são quebrados, como em dias de feriado, condições atmosféricas adversas, greves, festividades pontuais como concertos, comemorações regionais, Natal e passagens de ano. Na exploração aos dados não poderíamos deixar de verificar estas situações. Como amostra destas exceções à regra nos dados, decidimos avaliar dois dias muito particulares do ano de 2017: o dia 1 de Janeiro, o primeiro dia do ano, e o dia 8 de Fevereiro, feriado nacional que representa o Dia Esloveno da Cultura.

Na primeira amostra, ilustrada na Figura 3.15a, representa-se a média de ocupações de todos os Domingos do mês de Janeiro e onde o dia 1 se inclui. Podemos constatar que neste dia a ocupação foi inferior em quase todas as horas do dia em relação a todos os outros dias do mês, à exceção das 2, 3 e 4 horas da madrugada, resultado previsível devido às comemorações de Ano Novo. Na segunda amostra, Figura 3.15b, podemos observar a análise da ocupação da semana de 8 a 14 de Fevereiro. A ocupação média neste caso também diminui no dia 8 em relação aos outros dias com exclusão dos dias 11 e 12. Como o dia 8 de Fevereiro de 2017 foi uma quarta-feira, o dia 11 e 12 correspondem a Sábado e Domingo, respetivamente. Por conseguinte podemos então afirmar que a ocupação média nos feriados se assemelha à ocupação dos fins de semana num sensor no centro da capital.



(a) Média de ocupação de todos os Domingos de Janeiro 2017



(b) Média de ocupação da semana 8 a 14 Fevereiro 2017

Figura 3.15: Análise do dia de feriado nacional da Eslovénia (8 Fevereiro) e do primeiro dia do ano (1 de Janeiro) de 2017

Uma das análises mais importantes na exploração dos dados é a distinção no meio dos outros dados dos *outliers*, valores atípicos em português, que são valores que fogem ao comportamento normal dos dados tratados e que necessitam de ser identificados para que não distorçam futuramente a análise aos mesmos. Para descobrirmos estes valores iremos recorrer a um tipo de gráfico chamado *BoxPlot* (gráfico de caixa), método alternativo ao histograma e que fornece informações, para além da identificação de *outliers*, das seguintes características dos dados: localização, dispersão e assimetria. Além disso, este tipo de gráficos concedem um conhecimento mais abrangente da distribuição dos dados.

A análise vai ser realizada para a primeira faixa de rodagem para valores de velocidade e ocupação médias diárias por mês para todos os sensores. Não obstante, é de notar que para o caso da velocidade média não faz sentido traçar este gráfico para todos os sensores, porque os resultados seriam inconclusivos e portanto a análise será efetuada apenas a 10 sensores localizados nos principais acessos a Liubliana.

Comecemos com a interpretação do *BoxPlot* para o ano de 2016, Figura 3.16. As conclusões que podemos retirar quando analisamos um gráfico de caixa são: centro dos dados (mediana), a amplitude dos dados (max-min), a simetria do conjunto de dados e a presença de *outliers*. O retângulo contém 50% dos valores do conjunto de dados e a linha presente no mesmo representa o valor da mediana, sendo que a sua posição infere sobre a assimetria da distribuição. Em relação a esta característica no ano de 2016, a maioria dos meses possuem uma simetria negativa e apenas os meses de Julho e Setembro uma distribuição simétrica dos dados. Quanto à amplitude dos mesmos podemos concluir que o mês de Abril é aquele que se evidencia em relação a todos os outros.

Como já exposto, a identificação dos *outliers* é fundamental e estes aparecem como pontos ou asteriscos fora das "linhas" de máximo e mínimo desenhadas. Na Figura 3.16 podem ser identificados vários *outliers*, identificados a vermelho, divididos por 7 meses. No entanto, aqueles cujo número de valores atípicos é mais elevado são os meses de Junho, com 4 valores nos dias 5, 12, 19 e 26, e o mês de Setembro com 5 valores nos dias 4, 11, 18 e 25.

Como complemento a esta análise podemos afirmar que o dia com a média mais alta de ocupação em todos os sensores disponíveis em 2016 foi o dia 19 de Agosto e o dia com menor foi o dia 2 de Fevereiro, ressalvando apenas que apesar do dia 31 de Março ter uma ocupação menor, este não está completo como já referido aquando da análise da disponibilidade dos dados. É ainda de realçar que valores atípicos por defeito, como os de 25 de Dezembro e 1 Maio podem ser justificados com o dia de Natal e o feriado do Dia do Trabalhador, respetivamente, os restantes podem ser justificados por inúmeras razões como más condições atmosféricas, tolerâncias de ponto, períodos de férias escolares, entre outras.

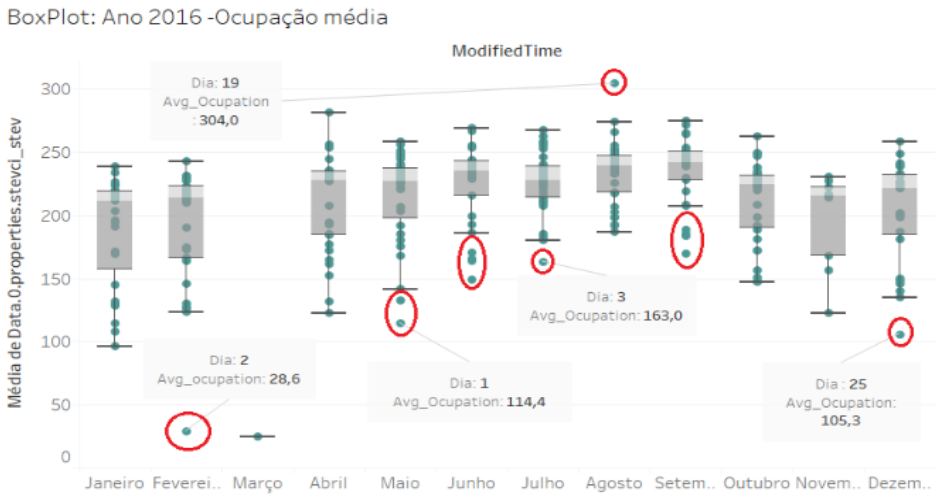


Figura 3.16: BoxPlot 2016 - Ocupação média

Relativamente aos primeiros cinco meses do ano de 2017, a interpretação é análoga com a particularidade de que não existem *outliers* em nenhum dos meses. Neste ano, o *BoxPlot* ilustrado na Figura 3.17, demonstra que novamente o mês com maior amplitude de dados é o Abril, tal como em 2016, e cuja mediana também é mais alta. Em relação à simetria, todos os meses apresentam uma distribuição simétrica negativa. Há ainda a referir, que o dia com menor ocupação durante os cinco meses foi o dia 1 Janeiro e o dia com maior ocupação o dia 9 de Maio, cujas anotações podem ser observadas na figura abaixo.

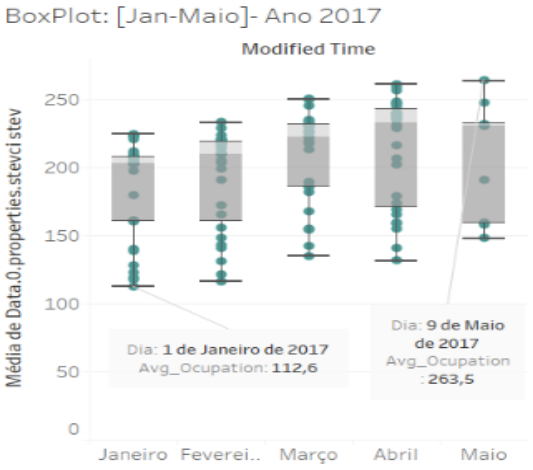


Figura 3.17: BoxPlot (Janeiro - Maio) 2017 - Ocupação média

No que diz respeito à velocidade média foi realizada a mesma análise, mas desta vez apenas a 10 sensores, como geograficamente demonstrado na Figura 3.18, pois avaliar a velocidade média em todos os sensores disponíveis não traria grande benefício. Isto deve-se ao facto de existirem sensores localizados em todo o tipo de estradas, com diferentes padrões de velocidade, isto é, uma autoestrada não tem o mesmo perfil de velocidade que uma estrada secundária por exemplo, e isso levaria a resultados inconclusivos. Pela mesma razão, para se poder retirar padrões de velocidade média foram também apenas consideradas 17 horas em cada dia, das 7 às 23 horas, para evitar horas em que o trânsito é quase nulo na maioria dos sensores. Os sensores escolhidos para objeto de estudo têm duas características principais: estão todos localizados na área circundante de Liubliana e localizados em autoestradas, como pode ser visto no mapa abaixo apresentado.

Figura 3.18: Localização geográfica dos sensores no acesso a Liubliana

Ao contrário de Portugal em que o limite de velocidade nas autoestradas é de 120 Km/h, na Eslovénia esse limite é de mais 10 km/h, ou seja 130 km/h. Podemos afirmar, dado a análise aos gráficos de caixa de 2016 e 2017, que os eslovenos são bastante cumpridores das regras no que que diz respeito a esta medida, relativa à média da mesma. Vejamos primeiramente esta análise para 2016, Figura 3.19. Em termos de valores atípicos de velocidade neste ano é importante referir que existe apenas três valores, distribuídos por três meses diferentes: Fevereiro, Setembro e Novembro. Em termos de simetria e amplitude temos os meses de Fevereiro e Maio com uma distribuição simétrica e o mês de Junho como o mês de maior amplitude média de velocidade de 2016. Em relação à premissa da primeira frase deste parágrafo, podemos verificar a sua veracidade ao observarmos que na maioria dos meses os valores de média de velocidade por dia rondam aproximadamente os 70 km/h.

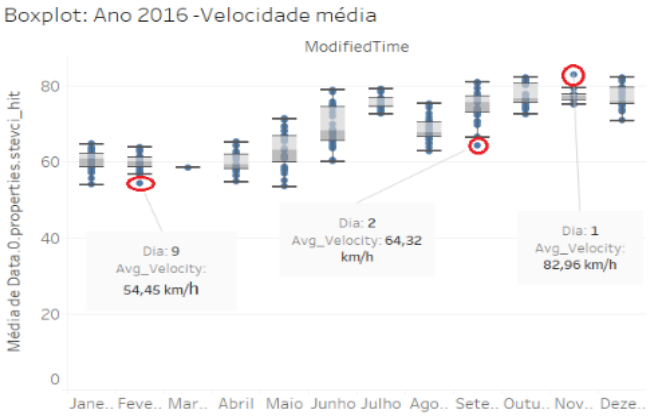


Figura 3.19: BoxPlot 2016 - Velocidade média

No ano de 2017 a situação altera-se um pouco, principalmente em relação à média de velocidades. No ano de 2016 completo, a média de velocidades é cerca de 70 km/h como já mencionado, mas se nos focarmos apenas nos primeiros cinco meses, essa característica desce para valores perto de 60 km/h. Desta forma, e como no ano de 2017 temos apenas cinco meses, podemos afirmar que em comparação com o mesmo período do ano precedente esse valor aumentou sensivelmente 20 km/h. Na Figura 3.20 pode observar-se esse fenómeno, e perceber que a amplitude dos dados em todos os meses é praticamente igual, registando-se apenas um *outlier* no dia 13 de Janeiro, anotado a vermelho na figura.

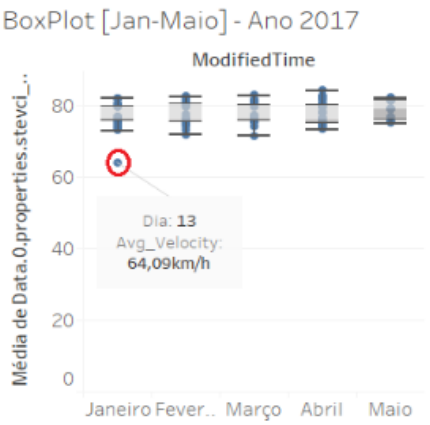


Figura 3.20: BoxPlot (Janeiro - Maio) 2017 - Velocidade média

Todas estas análises contribuíram para uma melhor compreensão dos dados. Em termos de qualidade, os dados apresentam um espectro muito vasto de informação para conseguirmos retirar várias conclusões na sua análise. Contrariamente foram identificadas situações em que a análise aos mesmos não produziu resultados fidedignos. Exemplos incluem a velocidade média que é calculada e não medida pelos sensores, descrições do estado do tráfego incorretas ou registos repetidos.

Em relação à disponibilidade dos dados, estes apresentam uma média de aproximadamente 71% em todos os 17 meses analisados. Apesar desta percentagem, existem meses como o de Março e Novembro de 2016 e Maio de 2017, cuja percentagem de disponibilidade apresenta valores abaixo de 30%, e que por essa razão não apresentam uma disponibilidade suficiente para uma análise profunda aos mesmos.

Na exploração dos dados emergiram facilmente alguns padrões como o de sazonalidade, registaram-se maiores ocupações durante o Verão, menores nos feriados, Ano Novo e fins de semana. Adicionalmente, surgiram alguns valores atípicos de ocupação e velocidade que não apresentam uma razão lógica e plausível e desse modo podem expor medições erradas dos sensores, avarias ou falta de manutenção nos mesmos. Em suma, os dados analisados apresentam qualidade e disponibilidade suficientemente razoável para serem tratados.

Capítulo 4

Implementação

Este capítulo é extremamente importante para perceber quais as tecnologias utilizadas e como estas se juntam e interagem para implementar o protótipo da aplicação de *real time analytics*, desde a sua arquitetura à forma como os dados fluem, desde a base de dados até ao *front end*. Resumidamente, visa descrever os detalhes de especificação e implementação do demonstrador desenvolvido.

4.1 Tecnologias

Esta secção explicita as principais tecnologias adotadas para desenvolver a aplicação em questão. É de sublinhar que todas as tecnologias usadas estão sob licença de código aberto (*open source*).

A base de dados utilizada nesta implementação foi o MongoDB, que serve para guardar os dados antes do processamento. O design e modelação dos dados no Mongo foi possível utilizando o Studio3T IDE, uma ferramenta muito completa que fornece suporte e apoio aos programadores que trabalham com o Mongo.

À exceção do *front-end* que utiliza linguagens de programação comummente designadas por linguagens de programação Web, toda a camada lógica da aplicação foi desenvolvida na linguagem de programação Java utilizando o Eclipse IDE e o IntelliJ IDE para o efeito, editores que fornecem um ambiente visual integrado para várias linguagens e paradigmas.

A comunicação entre a camada de processamento de dados e o *front end* é realizada por WebSockets, suportada pelo servidor Web Jetty. A interface com o utilizador foi implementada com as tecnologias HTML5, CSS 3 e JavaScript. Esta última possui o papel mais importante, pois é responsável pela implementação do cliente WebSocket que recebe os dados do Storm, bem como da interação utilizador/browser, através de animações ou

deteção de eventos.

Podemos observar na Tabela 4.1 as principais tecnologias utilizadas tal como a sua descrição abreviada.

Tabela 4.1: Principais tecnologias utilizadas

Java	A aplicação tem um suporte imenso na linguagem de programação Java. As topologias do Apache Storm são desenvolvidas inteiramente em Java, bem como a injeção dos dados provenientes do MongoDB para o RabbitMQ. Para além disso, toda a limpeza e seleção aos dados realizada anteriormente a esta implementação foi também realizada nesta linguagem, com o auxílio do MongoDB Java Driver.
RabbitMQ	RabbitMQ é um software de código aberto (*open source*) que foi implementado para suportar um protocolo de mensagens denominado Advanced Message Queuing Protocol (AMQP). Através desta plataforma, é possível criar uma aplicação para lidar com o tráfego de mensagens. Este aplicativo funciona num servidor e tem como ideia principal a disponibilização de uma estrutura que facilita fluxos de mensagens, sobretudo em grandes aplicações, para a comunicação entre todos os processos da mesma.
Jetty	Jetty fornece um servidor Web e um *container javax.servlet*, além de suporte para HTTP, WebSocket, OSGi, JMX, JNDI, JAAS e muitas outras integrações. Tem a capacidade, a partir da sua WebSocket API, de conectar os *endpoints* de WebSockets às especificações do caminho *servlet* através do uso de um *servlet* de *bridge*, que foi muito útil para a passagem de dados da aplicação para o *front-end*.
spring	O Spring é um framework *open source* para a plataforma Java baseado nos padrões de projeto inversão de controlo (IoC) e injeção de dependências. No Spring o *container*, encarrega-se de "instanciar" classes de uma aplicação Java e definir as dependências entre elas através de um arquivo de configuração em formato XML *auto-wiring*, ou ainda anotações nas classes, métodos e propriedades. Dessa forma, o Spring permite a conjunção de uma forma facilitada entre classes de uma aplicação orientada a objetos.
mongoDB	MongoDB é uma ideia recente de banco de dados que traz o conceito de Banco de Dados Orientado a documentos. É também chamado de banco de dados NoSQL. Tem como característica principal conter todas as informações importantes num único documento, livre de esquemas e tabelas. Assim, possibilita a consulta de documentos através de métodos avançados de agrupamento e filtragem.
ActiveMQ	ActiveMQ é um *message broker open source* escrito em Java. Um *broker* é um produto baseado no conceito MOM (Message Oriented Middleware) que precede à criação do Java Message Service (JMS). Consiste basicamente num sistema intermediário a outras aplicações, o qual recebe, envia e redireciona mensagens destas e para estas aplicações, suportando assim processos assíncronos e distribuídos tolerantes a falhas. É de realçar, que um *broker* possui as suas próprias interfaces de comunicação com os seus clientes e bem como suporte à API JMS.
Apache Camel	O Apache Camel é essencialmente um framework de integração. No núcleo da sua estrutura é um motor de roteamento. Este sistema permite definir regras próprias de roteamento como definir de quais fontes aceitar mensagens e determinar como processá-las e enviá-las para outros destinos.

A arquitetura desenvolvida, descrita na próxima secção, e que faz a ligação entre todas estas tecnologias, é baseada numa arquitetura com o nome de *Lambda Architecture* [16], desenhada por Nathan Marz, o fundador do Apache Storm. *Lambda Architecture* é uma framework útil e generalista para ajudar a desenhar aplicações que tratam um grande volume de dados. Esta arquitetura pode ser observada na Figura 4.1, todavia é de destacar que no âmbito deste trabalho estamos apenas interessados na parte da arquitetura que trata dados em *streaming*, isto é, na *Speed layer*.

[1] Fonte: Autor: Guido Schmutz, Título: Big Data und Fast Data - Lambda Architektur und deren Umsetzung

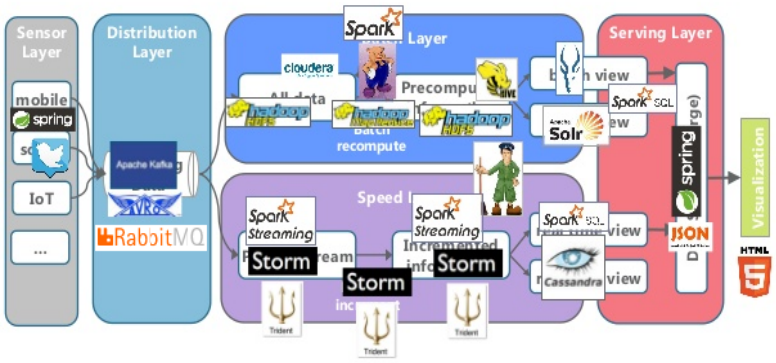


Figura 4.1: *Lambda Architecture*[1]

4.2 Arquitetura Desenvolvida

Em relação à arquitetura da aplicação podemos dividi-la em 3 fases fundamentais: ingestão e modelação de dados, de forma a corresponder aos requisitos da plataforma de processamento de *streams*; processamento através dessa plataforma, mais especificamente pelo Apache Storm, dos dados; e por fim visualização de dados, que através da ligação do output do Storm com o *front end*, confere "vida" aos dados, quer seja através de gráficos e/ou mapas. A arquitetura geral da aplicação pode ser observada na Figura 4.2.

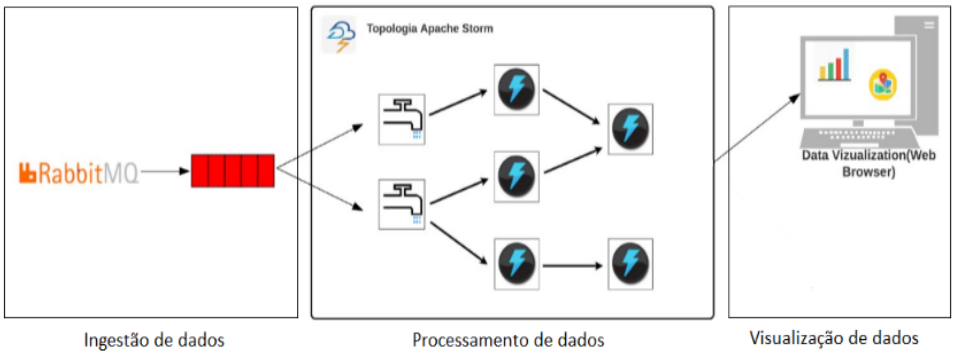


Figura 4.2: Arquitetura geral da aplicação

4.2.1 Ingestão e Modelação de Dados

O primeiro problema que surgiu no desenvolvimento do protótipo foi a ingestão dos dados na ferramenta de processamento de *streams*. Os dados que possuímos são estáticos e estão guardados na base de dados MongoDB. Do mesmo modo e sendo o objetivo simular uma aplicação em tempo real, estes dados teriam de ser inseridos na topologia do Storm num certo intervalo de tempo repetidamente para o efeito, como se tivessem a ser debitados no sistema em tempo real. Numa primeira fase da implementação desta ingestão, era a própria instância do *spout*, instância responsável pela ingestão de dados

na topologia do Storm, que fazia a conexão diretamente ao Mongo, o que se revelou imediatamente uma solução bastante vulnerável no que diz respeito a tolerância a falhas. De facto, se acontecesse algum problema ao servidor do MongoDB, a topologia perdia a conexão, e assim sendo interrompia o fluxo de dados e consequentemente o seu processamento. Além de tudo isto, é essencial que a fonte de dados seja confiável, ou seja, que suporte o reenvio de mensagens para a topologia em caso de falha. Tal não acontece com o Mongo, que quando envia uma mensagem para o *spout*, unidade do Storm responsável pela ingestão de dados, deixa de assumir responsabilidade sobre a mesma.

Por outro lado, uma fonte de dados confiável passa as mensagens para o *spout*, mas não assume automaticamente que a responsabilidade passe a ser da ferramenta de processamento. Em vez disso, espera até que esta lhe informe que recebeu a mensagem, caso esta confirmação não chegue à fonte de dados esta tem a capacidade de a enviar novamente mais tarde. Como já visto noutro Capítulo existem inúmeras fontes de dados com estas características, que são usualmente utilizadas com este sistema, Apache Storm. A escolha no caso da aplicação em questão para esta fonte de dados recaiu sobre o RabbitMQ, um *queuing system*. Este sistema utiliza o protocolo Advanced Message Queuing Protocol (AMQP), um protocolo de enfileiramento de mensagens que permite a troca de mensagens assíncronas entre diferentes aplicações e/ou serviços.

Foi desenvolvido então um programa em Java que escreve para uma fila (*queue*) com um determinado nome presente no servidor do RabbitMQ, executado localmente e que é depois consumida pelo *spout* na topologia. Pode ser realizada uma analogia com a Figura 4.3, onde o *producer* representa o programa em Java desenvolvido que retira os dados do Mongo e os *consumers* o paralelismo dos *spouts* no Storm que consomem os dados.

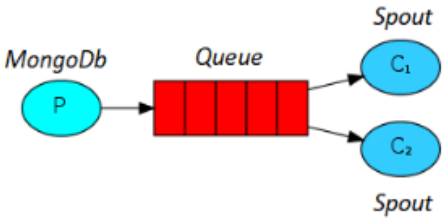


Figura 4.3: Arquitetura do sistema de *queues*

Vejamos então pormenorizadamente a lógica deste *producer* de mensagens. Este programa abre uma conexão à base de dados MongoDB onde estão armazenados os dados e realiza *querie*s pela próxima data de uma forma recorrente. Desta forma, a cada iteração desta *querie* o Mongo devolverá todos os documentos que correspondem a uma determinada data (dia, mês, ano, hora, minutos e segundos) e depois a próxima data e assim sucessivamente, sendo esta parametrização realizada pelo utilizador. Ao percorrermos estes documentos convertemos as suas informações para um objeto Java e que posteriormente enviaremos para a *queue*. Este objeto a ser enviado possui dois atributos, a data a que foram realizados os registos e uma lista contendo todos os sensores e suas respetivas leituras de velocidade, ocupação, estado de tráfego e diferença de tempo entre veículos. É

de realçar que cada sensor é distinguido pelo seu *Id* e que antes de ser enviado o objeto para a fila do RabbitMQ este é convertido para uma string de formato JSON através da biblioteca Gson, uma biblioteca Java que pode ser usada para converter objetos Java na sua representação JSON e vice-versa. Para ilustrar todo o processo de ingestão de dados, vamos utilizar um exemplo apenas com dois sensores, que pode ser observado na Figura 4.4. É de destacar que o processo com a quantidade real de sensores que existem por mês é idêntico.

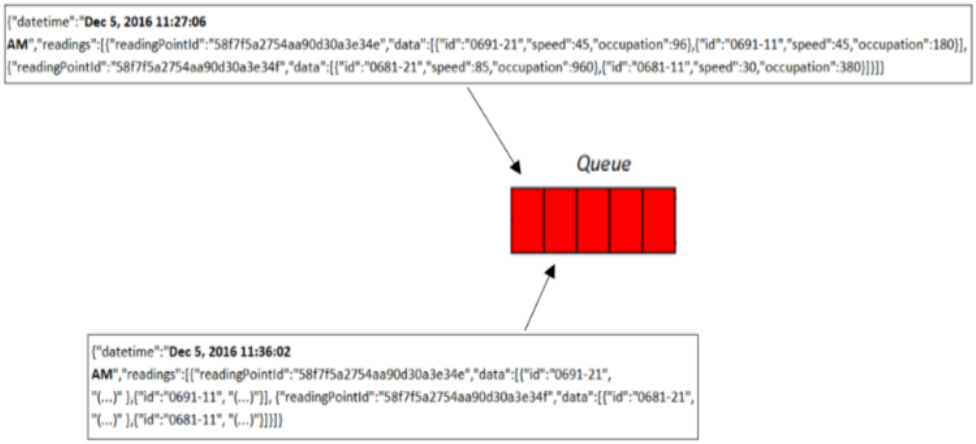


Figura 4.4: Processo de inserção de dados na fila de mensagens do RabbitMQ

Uma pergunta muito pertinente e que pode surgir ao observarmos a Figura 4.3 é como o RabbitMQ entrega as mensagens aos *spouts* tendo em conta o paralelismo do Storm. Esta ferramenta de processamento de *streams* tem a capacidade de realizar vários processos (*threads*) em simultâneo da sua topologia e pode ser executada em *cluster* de forma a aumentar facilmente a sua capacidade de computação.

Imaginemos o caso em que temos um paralelismo de 2, ou seja, 2 *threads* a serem executadas simultaneamente no *cluster*, no que diz respeito ao número de *spouts* a consumir dados da fila do RabbitMQ. Uma das características principais deste sistema de mensagens é que distribui em média as mensagens equitativamente pelos consumidores. Consideremos que o trabalho de um *spout* é muito mais intenso do que o do outro, o RabbitMQ não sabe deste facto e vai continuar a mandar mensagens da mesma maneira, o que leva a que um *spout* esteja a realizar um grande trabalho e outro quase nada, o que pode trazer problemas para a topologia no futuro. Porém, é possível a configuração do mesmo para entregar apenas uma mensagem de cada vez após a confirmação do *spout* do processamento e receção da anterior, indicando que está pronto para receber a próxima mensagem, o que torna o processamento muito mais equilibrado, sendo que foi este o procedimento tomado.

Como já mencionado, o RabbitMQ é considerado uma fonte de dados confiável, ou seja, é um sistema tolerante a falhas [34]. Analisemos o seu comportamento na pior situação. Suponhamos que um *worker*, nó do *cluster* do Storm, onde é executado o *spout* é retirado desse mesmo *cluster* com uma topologia em funcionamento. O que acontece às

mensagens que estavam a ser processadas? São perdidas? O RabbitMQ tem um sistema de confirmação de mensagens tal como o Storm, isto é, se o *spout* não tiver enviado o *ack*, a indicar que recebeu a mensagem, o mesmo entende que a mensagem não foi processada e reenvia-a para outro consumidor (*spout*) que esteja disponível. Caso contrário, espera que o Nimbus, *master node* do *cluster* responsável pela distribuição de coordenação da topologia, redistribua o *spout* em questão por outro *worker* para lhe entregar a mensagem, só neste ponto a mensagem é apagada da fila de receção de mensagens do Rabbit. Desta forma, temos um sistema robusto de entrega de mensagens e que consegue reagir a falhas que possam ocorrer durante todo o processo.

4.2.2 Processamento de Dados

A 2.ª fase da arquitetura, e a mais importante, corresponde ao efetivo processamento de *streams* em tempo real através da ferramenta Apache Storm. É durante esta fase que são transformadas as informações provenientes da base de dados para se retirar alguma informação em tempo real sobre as mesmas. Estas informações podem ser guardadas novamente na mesma ou observadas através de técnicas de visualização em tempo real na última fase do projeto. De seguida, iremos observar a configuração pormenorizada das duas topologias desenvolvidas.

4.2.2.1 Topologias Apache Storm

Foram desenvolvidas duas topologias no Apache Storm que realizam médias de valores de velocidade e ocupação por direção, porém de forma muito diferente. Apesar da topologia das duas ser muito semelhante, o resultado e a forma como as instâncias dos nós de processamento interagem entre si difere ligeiramente. A primeira topologia tem como objetivo o cálculo da média dos valores de velocidade e ocupação por direção para todas as faixas. Desta forma, se numa direção temos duas faixas, o output em tempo real desta topologia será a média de velocidade e ocupação nas duas faixas dessa direção numa determinada data e hora. Como temos duas direções, teremos quatro outputs de média instantânea, duas de ocupação e duas de velocidade. Vejamos o *design* geral da topologia desenvolvida com os respetivos nomes das classes que implementam os *spouts/bolts* da mesma na Figura 4.5.

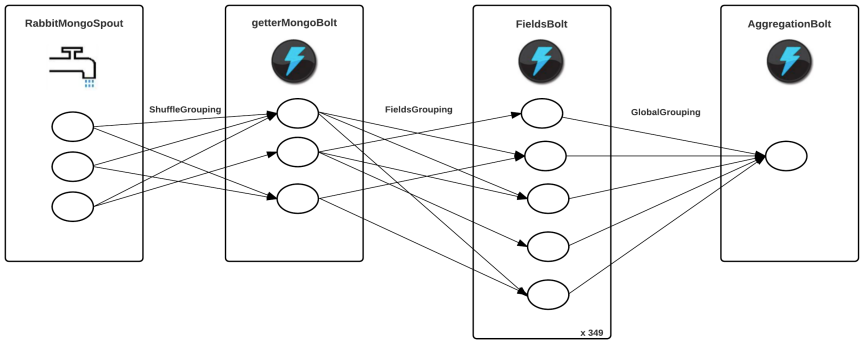


Figura 4.5: Arquitetura da topologia do Apache Storm

Os *spouts* no Storm têm a função de inserirem novos dados para a topologia, e neste caso o *RabbitMongoSpout* não foge à regra. Este *spout* é responsável por consumir as mensagens presentes na *queue* do RabbitMQ. Para este efeito, verifica se existem mensagens novas a cada 1ms na *queue*, tempo limite imposto, pois a função onde é executado o consumo de mensagens não é permitida bloquear, por uma questão de segurança das *threads* no Storm [38], sob pena de paragem do fluxo de dados, caso hajam absorve-as para a topologia. Para além disso, converte a string no formato JSON proveniente da mesma em objetos Java para serem enviados para o *bolt* da instância seguinte. Isto significa que os objetos Java existentes no programa desenvolvido para inserção na fila são os mesmos que existem na topologia existente. Desta forma, é como um processo de *serialization* e *deserialization* de um *stream* de *bytes* entre o servidor do RabbitMQ e o *spout* do Apache Storm.

Seguidamente, o objeto é passado para o *bolt getterMongoBolt*, numa ligação de *ShuffleGrouping*, isto é, os *tuples* são passados entre instâncias da topologia de uma forma aleatória. É neste *bolt* que é realizada uma operação muito importante e que resume a forma como realizamos o processo. De facto, o objeto em Java contém a data e uma lista de sensores com as suas respetivas velocidades e ocupações durante esse período, mas para podermos realizar médias de forma paralela temos de dividir os sensores por *Id* para a próxima instância de *bolts*, e esta é a funcionalidade principal desta unidade de processamento. Desta forma, é emitido para a próxima instância de *bolts* através do *FieldsGrouping*, *grouping* que permite a partição de *tuples* por campo, neste caso o campo *Id*, a informação de velocidade, ocupação, data e hora, e *Id* por sensor. Por este motivo, cada instância/*thread* do *bolt FieldsBolt* corresponde à informação de um único sensor e tem a função de computar médias de velocidade e ocupação das várias faixas por direção, como mencionado anteriormente. Por fim, estes *tuples* são agregados através do *globalGrouping*, que envia todos os *tuples* para um único *bolt*, neste caso o *bolt aggregationBolt*, para serem posteriormente inseridos numa base de dados ou enviados para o browser para alguma forma de visualização de dados.

A segunda topologia desenvolvida também computa médias, mas desta vez horárias. A arquitetura geral é a mesma, porém com uma pequena diferença na forma como os sensores são divididos para o *FieldsBolt*. É fulcral entendermos como o *FieldsGrouping* no

Storm funciona para posteriormente percebermos porque este campo foi alterado nesta topologia. Este *grouping* é calculado da seguinte forma: *hash(fields)%(n.º tasks)*, sendo *hash* uma função de *hashing* do Storm. A forma como este algoritmo foi desenhado não garante que cada tarefa receba *tuples* para processar. A título de exemplo, se aplicarmos o *FieldsGrouping* num campo suponhamos *X*, com apenas dois valores possíveis *A* e *B*, e com um paralelismo de 2 para a instância do *bolt* para onde irão os *tuples*, pode acontecer que *hash(A)%2* e *hash(B)%2* sejam iguais, e por essa razão o resultado pode ser a ida de todos os *tuples* para uma única tarefa, enquanto a outra tarefa fica completamente inativa.

Na primeira topologia este problema não é tão evidente, pois o valor é calculado, enviado para a próxima instância e apagado para um futuro cálculo, que pode ser de outro *Id* no mesmo *bolt*, isto é, não tem de manter uma variável a guardar um determinado valor, à espera que um outro *tuple* chegue à tarefa (*bolt*) de processamento para a atualizar. Todavia, nesta segunda topologia, como referido, o objetivo é realizar médias horárias, ou seja, no exemplo de recebimento de dados de 10 em 10 minutos, podemos ter até um máximo de 6 *tuples* por hora num só *bolt* a atualizar a média a cada *tuple* que chega, pelo que garantir que um certo *Id* vai sempre para a mesma instância de *bolts* é fulcral sob pena de valores errados de cálculo. Só após ter passado uma hora é que a variável que computa o cálculo é apagada e passado o *tuple* correspondente para a última fase da topologia. Por todos estes argumentos foi indispensável a implementação de um *grouping* específico, não disponível pelo Storm, o *CustomGrouping*. Esta forma de ligação de instâncias específica garante, a partir da lista de inteiros contendo a numeração de todas as tarefas do Storm, que um determinado *Id* é direcionado sempre para uma dessas tarefas, que é exatamente isso que se pretende. Em termos de fluxo de mensagens esta topologia também se altera um pouco, enquanto que na primeira topologia o fluxo é contínuo, isto é, nunca há uma paragem no fluxo de dados, neste caso não é bem assim. O último *bolt* da topologia só recebe os dados do *bolt* que o precede quando a média horária estiver computada, no máximo este *bolt* só receberá dados após 12 registos terem sido consumidos, divididos por *Id* e processados para todos os sensores, valor máximo de referência para registos de 5 em 5 minutos.

Uma das características fundamentais do Apache Storm é a facilidade de paralelizar a topologia. Porém na forma como esta arquitetura está desenhada é de notar que existem dois nós que não podem ser paralelizados mais do que já estão, e para a explicação que se segue é necessário tomar atenção ao número que consta no canto inferior direito do *bolt FieldsBolt* da Figura 4.5. O número 349 representa o número de *threads* a executar paralelamente no *bolt* em questão e esse número corresponde igualmente ao número de sensores máximo por mês, ou seja, se os *tuples* estão a ser filtrados para esta instância pelo campo *Id*, uma topologia com mais *threads* a executar não iria resultar numa melhor velocidade de processamento, pois teria *threads* que não estariam a executar por não receberem dados. A outra instância que não pode ser paralelizada por ser uma instância de agregação é o último *bolt* da topologia. Todas os outros nós na topologia podem ser paralelizados, apenas tendo como limitação o número de *cores* do processador da máquina

em que a topologia está a ser executada.

4.2.3 Visualização de Dados

Esta sub-secção descreve a 3.ª fase do processo, isto é, o desenvolvimento de uma interface Web que permite a visualização de um *stream* de dados em tempo real processados via Apache Storm. A palavra visualizar no contexto computacional é a conversão de números ou categorias para um formato gráfico que pode ser facilmente compreendido [50] e é este o principal objetivo desta componente desenvolvida. Como já referido, esta visualização pode ser muito bem vista como uma simulação do que poderia ser uma aplicação em tempo real para monitorização de tráfego na Eslovénia.

Um dos principais requisitos desta integração, sendo esta uma aplicação de *streaming* em tempo real, é a latência do processo de envio dos dados entre a ferramenta Storm e o *front end*. Por esta última razão, a tecnologia utilizada para a comunicação foram os WebSockets. Recentemente o aparecimento de aplicações em tempo real tem vindo a aumentar, sites como Instagram, Facebook, jogos online, compra e venda de ações, chats, entre outros, utilizam a comunicação em tempo real. Mas o que é verdadeiramente uma aplicação em tempo real? Não é nada mais que uma aplicação na qual a informação é compartilhada em tempo real, sendo por isso necessário que o servidor seja capaz de informar todos os clientes conectados assim que uma informação chega ou no menor tempo possível, sem que estes tenham de fazer uma requisição para obtê-la. Antes de procedermos às tecnologias que surgiram para colmatar esta nova geração de aplicações vejamos como funciona resumidamente o processo de abertura de uma página Web.

Para que tenhamos algum conteúdo no nosso browser, temos que fazer uma requisição ao servidor, isto é, um pedido ao mesmo. Ao receber esse pedido o servidor processa a solicitação e responde mostrando o conteúdo da página para o cliente num protocolo com o nome de *HTTP*, entenda-se cliente o computador pessoal e servidor o computador onde estão hospedados os sites com as suas respetivas informações, imagens, textos, etc. Note-se que este é o procedimento usual realizado por vários sites na Web e que o processo é muito mais complexo, serve apenas para ilustração da forma como é realizado. Como podemos concluir, este tipo de interação cliente/servidor não funciona para aplicações em tempo real onde o requisito principal é a velocidade. As técnicas que surgiram, conhecidas como técnicas de *Push*, permitiram uma mudança de paradigma, ou seja, permitiram que o servidor enviasse dados para o cliente ao mesmo tempo que constata que novos dados estão disponíveis, acabando assim com a direcionada comunicação ao cliente que o *HTTP* tradicional impunha.

Uma das primeiras alternativas para tentar resolver o problema da comunicação em tempo real foi o *HTTP Polling*, em que requisições são enviadas do cliente ao servidor periodicamente. Após a resposta, a conexão cliente/servidor é fechada. Esta primeira abordagem traz inúmeras desvantagens, como perda de informação caso o tempo de

atualização não seja suficientemente pequeno, sobrecarga do servidor em busca de atualizações mais confiáveis, e por fim comunicação lenta devido ao cliente fazer requisições periódicas independentemente do servidor possuir ou não novas mensagens. Outra das tecnologias que surgiu foi a tecnologia *HTTP Long Polling* que é uma variação da primeira técnica. Neste caso, o cliente cria uma conexão com o servidor, que permanece aberta até que existam novas mensagens, ou seja, caso não hajam mensagens novas disponíveis a conexão é mantida, caso contrário são enviadas as mensagens e esperadas novas requisições por parte do cliente. A última técnica que surgiu é denominada de *HTTP Streaming*, que é considerada a melhor técnica assente no protocolo *HTTP* para comunicação em tempo real e concorrente direta da comunicação por WebSockets. Nesta técnica, o servidor mantém a conexão aberta e ativa com o cliente até que os dados necessários sejam recebidos, mantendo assim a conexão indefinidamente aberta. Tem uma desvantagem clara, pois ao incluir os *HTTP Headers* nos pacotes de dados trocados entre o servidor e o cliente aumenta substancialmente o tamanho dos arquivos e consequentemente atrasa o processo de transporte de dados.

Em todas estas técnicas existe uma requisição do cliente ao servidor e uma resposta do mesmo numa comunicação conhecida como *half-duplex*. Apesar de existirem várias outras técnicas tal como estas apresentadas compartilham um problema: carregam a sobrecarga do protocolo *HTTP* que não é adequada para aplicativos que requerem uma baixa latência, principalmente quando comparadas com a comunicação por WebSockets como conclui o estudo realizado por Victoria Pimentel [32].

WebSockets, tal como *HTTP*, é uma camada de transporte sobre a comunicação Transmission Control Protocol (TCP). A diferença entre os dois é que no clássico protocolo *HTTP* a resposta do servidor ao pedido do cliente fecha o *socket* TCP, enquanto que no protocolo WebSocket a conexão permanece aberta. Desta forma, permite que este protocolo apresente uma característica diferenciadora em relação a todas as outras que é a sua comunicação bi-direcional, ou *full duplex*, isto é, que permite que as informações fluam do servidor para o cliente e vice-versa. Uma conexão deste protocolo começa com um *HTTP GET request* do cliente para o servidor. Após a resposta do servidor numa ação denominada de *handshake*, ou em português "aperto de mão", cada uma das partes é livre de enviar mensagens, cuja abstração são *frames* de WebSockets. Por todas estas razões, este protocolo foi o utilizado para a transmissão dos dados provenientes do processamento no Apache Storm para o Web browser.

Existem outras mudanças e tecnologias envolvidas nesta fase da arquitetura. Uma das mudanças tem a ver com alterações realizadas à topologia do Apache Storm para dotar a aplicação de uma visualização em tempo real. A alteração na topologia deve-se à adição de um *bolt* como ilustrado na Figura 4.6.

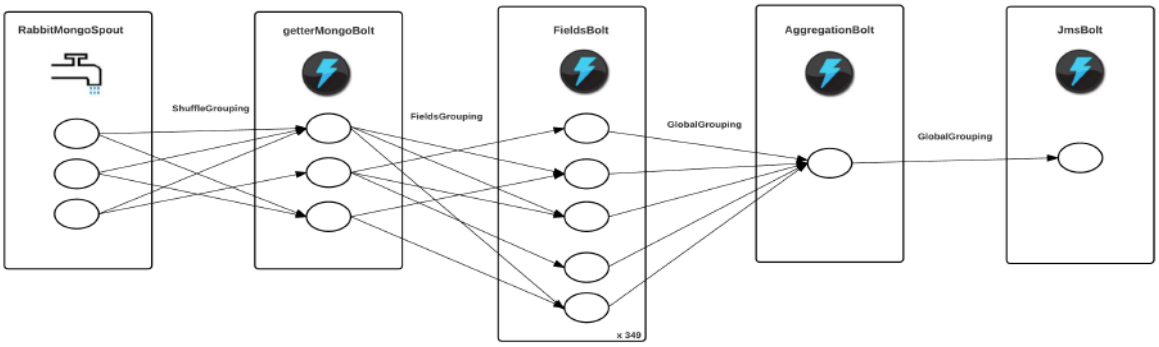


Figura 4.6: Arquitetura da topologia Storm para visualização em tempo real

A função deste *bolt* com o nome de *JmsBolt* é escrever para uma *queue* do ActiveMQ, um sistema similar ao RabbitMQ, a cada vez que as médias estiverem calculadas para todos os sensores e sejam enviadas através do *bolt* da instância anterior, o *AggregationBolt*, para o mesmo. A implementação deste *bolt* é fornecida por uma biblioteca de contribuições do Apache Storm com o nome de *storm-jms*, disponível em `https://github.com/ptgoetz/storm-jms`.

O ActiveMQ tal como o RabbitMQ é um *message broker open source* cujo objetivo é enviar mensagens entre dois aplicativos. Este sistema é escrito em Java e tem um cliente completo de JMS, uma Application Programming Interface (API) para middleware do Java orientado às mensagens. Esta API permite enviar mensagens entre dois ou mais clientes. É um sistema padrão de mensagens que permite que aplicativos baseados no Java EE (Java Enterprise Edition) criem, enviem, recebam e leiam mensagens. Um aplicativo pode comunicar com um qualquer número de aplicativos utilizando o JMS, enviando e retirando mensagens de um mesmo destino. A API JMS suporta dois tipos de modelos de mensagens: o modelo "ponto a ponto" e o modelo *publish/subscribe*. O modelo escolhido foi o "modelo ponto a ponto", ou por filas, que se caracteriza pela presença de um *producer* que envia mensagens e a de um *consumer* que as recebe. Neste caso, o produtor conhece o destino da mensagem e envia-a diretamente para a fila do consumidor, como pode ser observado na Figura 4.7.

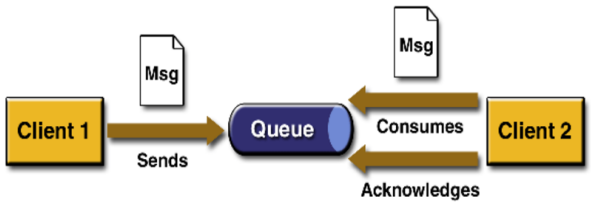


Figura 4.7: Modelo de troca de dados entre clientes JMS

Para se integrar facilmente o Storm ao *front end*, através da comunicação por WebSockets, foi escolhido o framework Apache Camel que faz todo o trabalho pesado. Este sistema é capaz de definir regras de *routing* e mediação num conjunto de linguagens de programação através de um *endpoint*, utilizando a implementação do WebServer WebSocket Jetty. Utiliza URLs para trabalhar diretamente com qualquer tipo de modelo de transporte ou mensagens, como *HTTP*, ActiveMQ, JMS, JBI, SCA, MINA ou CXF, bem como com diferentes formatos de dados. Assim sendo, ao termos o *bolt* na topologia a escrever o seu output para uma *queue* do ActiveMQ, foi criada uma *camel route* que se inscreve como consumidor desta fila e envia as mensagens em *frames* de WebSockets para todos os clientes conectados a essa *route*. Por este motivo, podemos ter várias páginas Web que recebem os mesmos dados e por isso podem ter cada uma a sua forma de apresentação de dados simultaneamente. Através da implementação de um cliente de WebSocket em JavaScript é possível obter a informação pretendida proveniente do processamento do Apache Storm. A arquitetura desta interação pode ser observada na Figura 4.8. É de sublinhar que uma *camel exchange* presente na figura é a abstração do Apache Camel para a troca de mensagens através de WebSockets.

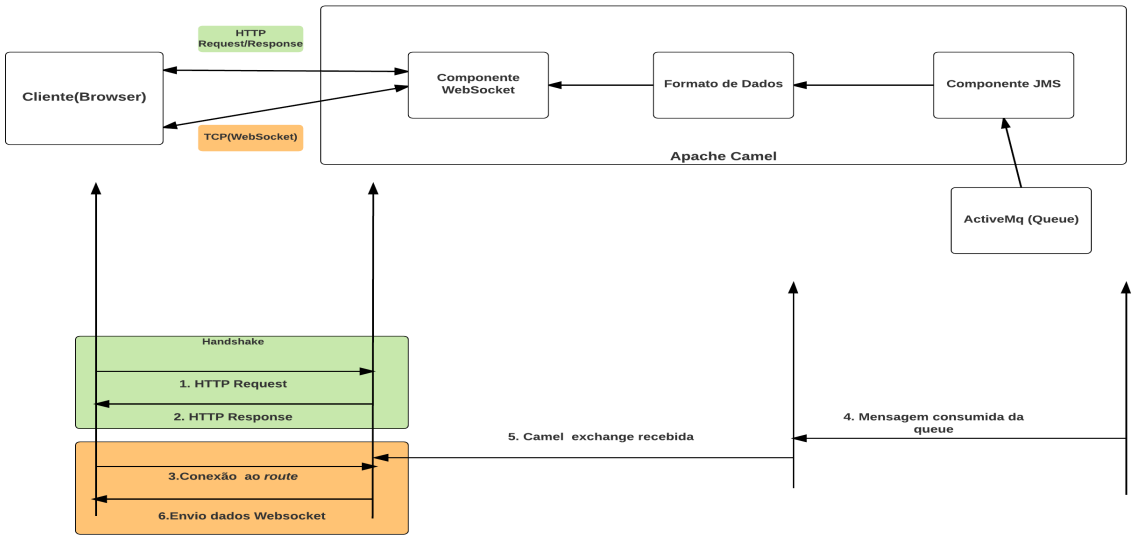


Figura 4.8: Arquitetura da ligação Apache Storm - Browser: Apache Camel[2]

[2]Fonte: Adaptado de: Título: Develop Real Time HTML5 Applications using WebSocket with Apache Camel and ActiveMQ, Autor: Charles Moulliard

4.3 Workflow da Aplicação

Depois da explicação realizada acerca da arquitetura, é essencial perceber-se como os dados percorrem as várias fases da mesma até poderem ser visualizados no browser. Todas as fases e transformações dos mesmos durante todo o processo será explicitado de seguida e ilustrado na Figura 4.9:

1. Os dados são retirados do MongoDB por ordem ascendente de data e hora e convertidos em objetos Java. O objeto a ser enviado contém data e hora, e uma lista com todos os sensores e suas velocidades e ocupações;

2. Para poderem ser enviados para a fila no servidor RabbitMQ, estes objetos Java são convertidos numa string com o formato JSON;

3. Esta fila é consumida periodicamente pelo *spout* do Apache Storm, responsável pela inserção e desconversão da string JSON em objetos Java novamente;

4. Este objeto é dividido em vários *tuples* pelo campo *Id* para poder calcular médias de velocidade e ocupação de forma paralela na topologia;

5. Todos os objetos são agrupados no último *bolt* da topologia após o cálculo;

6. São passados todos estes *tuples* para um *bolt* adicional que tem a função de novamente converter os objetos numa string Java, e inseri-los noutro sistema de mensagens em fila, mas desta vez o ActiveMQ;

7. O Apache Camel entra em ação subscrevendo-se à fila e emitindo as informações via comunicação WebSockets para todos os clientes conectados à sua *route*;

8. Por fim, através de um *parser* do JavaScript a mensagem e os seus respetivos valores são identificados para serem representados em diferentes visualizações como mapas e/ou gráficos no browser.

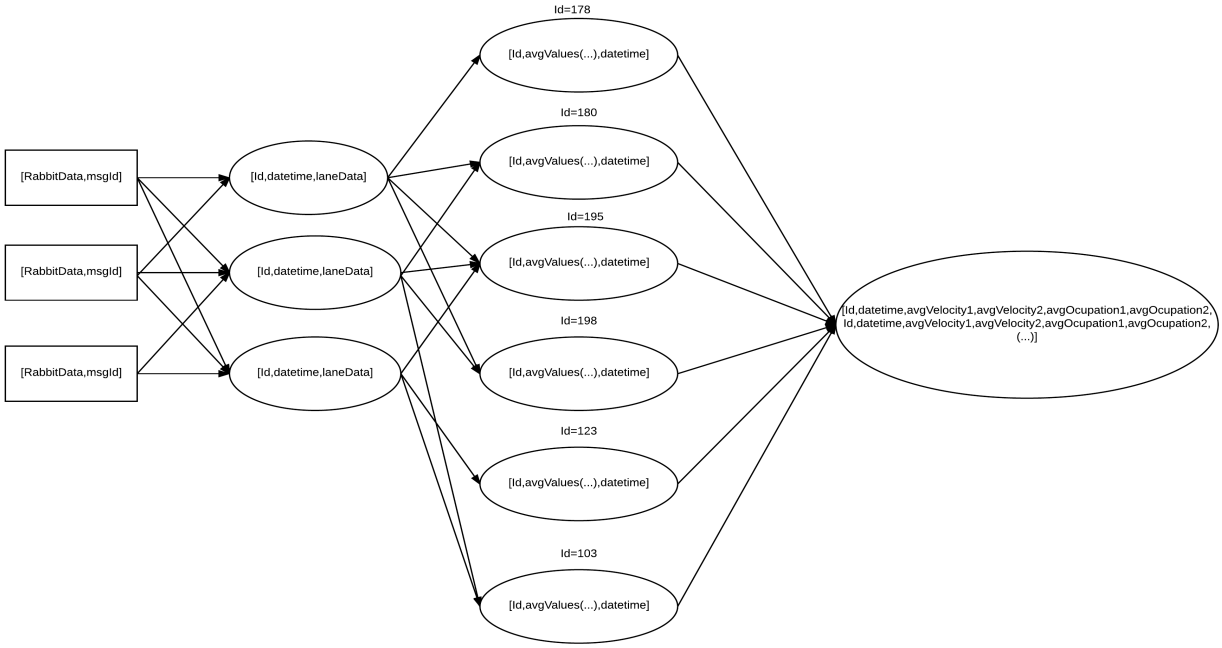


Figura 4.9: Workflow topologia Apache Storm

Capítulo 5

Validação e Resultados

Este capítulo descreve o método utilizado para avaliar a arquitetura proposta no capítulo anterior, nomeadamente a velocidade de inserção de dados na *queue* do RabbitMQ e as topologias do Apache Storm desenvolvidas, uma avaliação empírica para examinar a eficiência no processo de processamento em tempo real dos dados, desde a sua origem até à saída do processamento. Para além disso, serão demonstrados os resultados da visualização da aplicação em execução em tempo real bem como toda a discussão sobre os mesmos e também sobre os anteriores testes mencionados.

5.1 Metodologia de Teste

Antes de se descrever a metodologia utilizada para a validação, é preciso apresentar as características da máquina/computador onde os testes foram realizados, pois esta pode representar uma limitação aos resultados, e também dar a conhecer que programas foram utilizados.

Os testes foram realizados num computador com 4 núcleos, Intel(R) Core(TM) i7-7500U CPU @ 2,70Ghz, uma memória RAM de 8GB e um disco SSD de 256 GB, e os programas utilizados e suas versões foram: o cliente Apache Storm 0.9.1 - incubating, MongoDB 3.0.4 e RabbitMQ 3.6.10. O principal objetivo dos testes é pôr o Storm à prova no que diz respeito à velocidade de processamento. No entanto, tal como já explicado em capítulos anteriores, esta depende diretamente da base de dados e do sistema de *queues* referidos, pois é a partir de ambos que é realizado o processo de ingestão de dados na ferramenta de processamento de *streams*. O Storm tem dois modos de operação: modo local e modo remoto. No modo local, podemos desenvolver e testar topologias localmente, modo muito utilizado na fase de desenvolvimento do trabalho ao passo que no modo remoto, enviamos topologias para execução em *cluster*. Este último modo é o modo de teste que foi utilizado.

Os experimentos foram realizados com diferentes configurações, de forma a se poder retirar conclusões confiáveis a nível de latência em todo o processo de ingestão e processamento de dados. A topologia utilizada nos testes foi a ilustrada na Figura 5.1, cuja finalidade é a computação de médias horárias de velocidade e ocupação de veículos das várias estradas da Eslovénia.

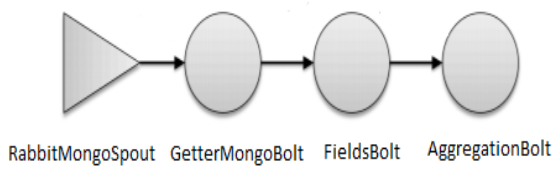


Figura 5.1: Topologia utilizada para os testes de validação da topologia Storm

Para estes testes foi utilizada apenas uma máquina física, com as especificações já mencionadas acima. A Tabela 5.1 mostra o número de tarefas utilizadas em cada instância de processamento na topologia. Foram realizados 2 testes diferentes, divergindo entre eles o número de *workers*, mais concretamente 1 e 4 *workers*, sendo este último o número máximo de *workers* possíveis no computador de teste, devido ao seu número de núcleos/*cores*.

Tabela 5.1: Parametrização da topologia a ser testada

Componente	# tasks
RabbitMongoSpout	200
GetterMongoBolt	200
FieldsBolt	350
AggregationBolt	1

Desta forma o *setup* de teste tem as seguintes características:

- 1 *master node*, o Nimbus;
- 1 ZooKeeper;
- 1 ou 4 *workers* executando independentemente cada um uma JVM diferente;
- 1 ou 4 *supervisor nodes* dependendo do número de *workers* utilizados;
- Número de *executors*/*threads* igual ao número de tarefas para cada instância de processamento, no total 751 *executors* + 1 *thread* por *worker*, *thread* responsável pelo *acknowledgement* no Storm.

É de sublinhar que o Nimbus, no caso da configuração com 4 *workers*, dividirá o número de *threads* equitativamente pelo número de *workers*, isto é, cada *worker* ficará responsável pela gestão de aproximadamente $\frac{751}{4}$ *threads*.

5.2 Resultados

Esta secção é dedicada aos resultados obtidos da execução da aplicação em tempo real tanto a nível de performance, no que diz respeito ao tempo de execução da ferramenta de processamento de *streams* e da ferramenta de ingestão de dados no mesmo (o RabbitMQ), como ao nível de output da mesma no browser através de várias formas de visualização de dados.

5.2.1 Resultados dos testes referentes à Ingestão e ao Processamento de *streams* em tempo real

Os resultados dos testes obtidos só foram possíveis devido ao facto de tanto o Apache Storm como o RabbitMQ possuírem uma API para gestão e monitorização da sua execução através de uma interface no browser. Obtiveram-se valores médios do número de mensagens por segundo na *queue* do RabbitMQ, velocidade de cada nó de processamento na topologia Storm e tempo total de toda a execução para as duas configurações. Vejamos de seguida os resultados obtidos:

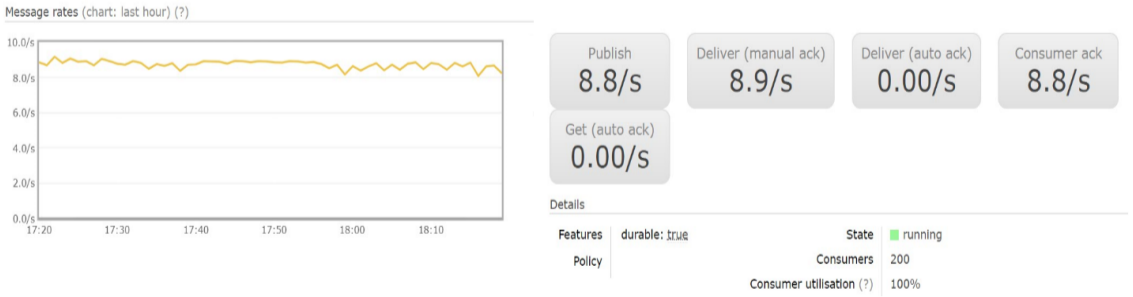


Figura 5.2: Número médio de mensagens inseridas na fila do RabbitMQ para configuração com 1 *worker*

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed
fields-bolt	350	350	2363140	2363140	0,086	0,034	2462220
getter-bolt	200	200	2462540	2462540	0,002	1,422	6960
tuples-Hour	1	1	0	0	0,075	0,019	2362840

Figura 5.3: Tempos de execução médios para as unidades de processamento do Apache Storm para configuração com 1 *worker*

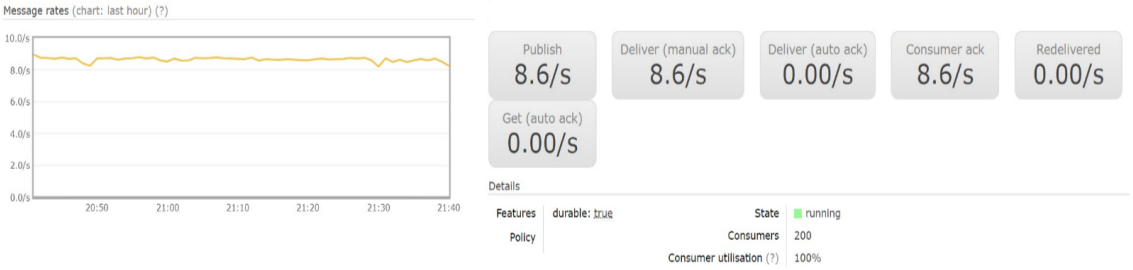


Figura 5.4: Número médio de mensagens inseridas na fila do RabbitMQ para configuração com 4 *workers*

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed
fields-bolt	350	350	605600	605600	0,002	0,022	1543080
getter-bolt	200	200	1542700	1542700	0,002	1,424	6080
tuples-Hour	1	1	0	0	0,014	0,021	604720

Figura 5.5: Tempos de execução médios para todas as unidades de processamento do Apache Storm para configuração com 4 *workers*

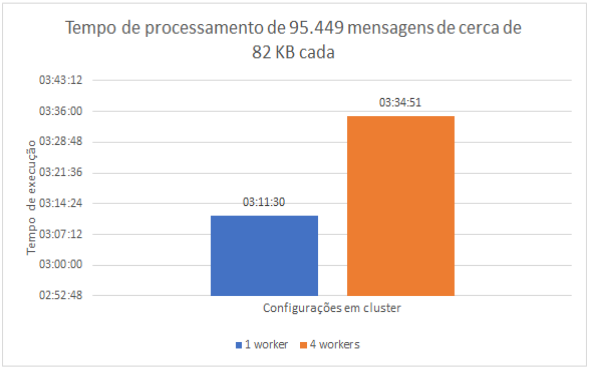


Figura 5.6: Tempo de execução para processar todos os dados dos 17 meses

5.2.2 Resultados da Visualização de Dados

Apesar das imagens que vão ser apresentadas corresponderem ao output da aplicação no browser, as mesmas não acrescentam grande valor no sentido da perceção real da execução da aplicação. De facto, todas estas imagens podem ser consideradas como *frames* de um vídeo, em que vão sendo alteradas conforme os dados vão saindo do Apache Storm, contudo é possível ter uma conceção das várias formas de visualização desenvolvidas.

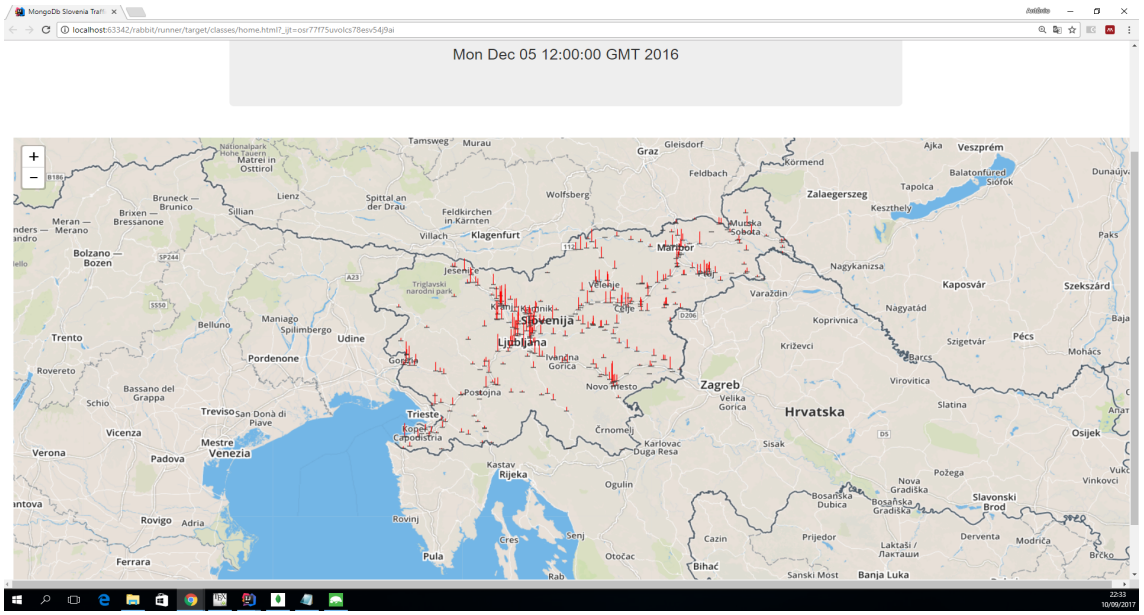


Figura 5.7: Visualização da ocupação média horária das estradas na Eslovénia a partir da elevação de barras localizadas nos sensores num mapa, utilizando a API *Leaflet*

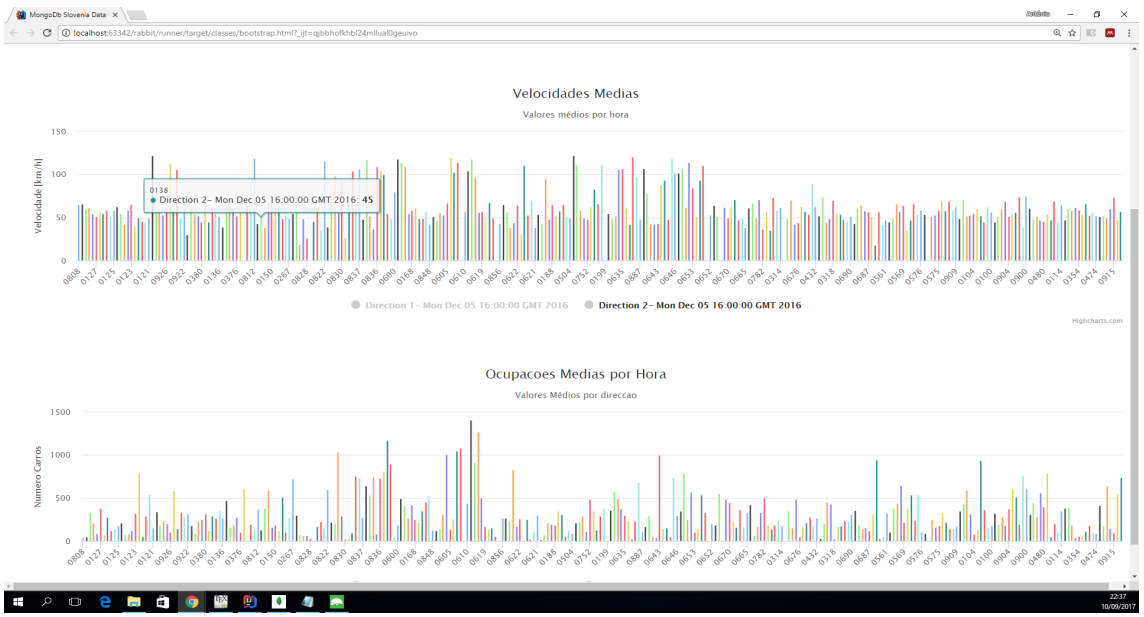


Figura 5.8: Visualização da ocupação e velocidade por direção e por sensor, utilizando a API *Highcharts*

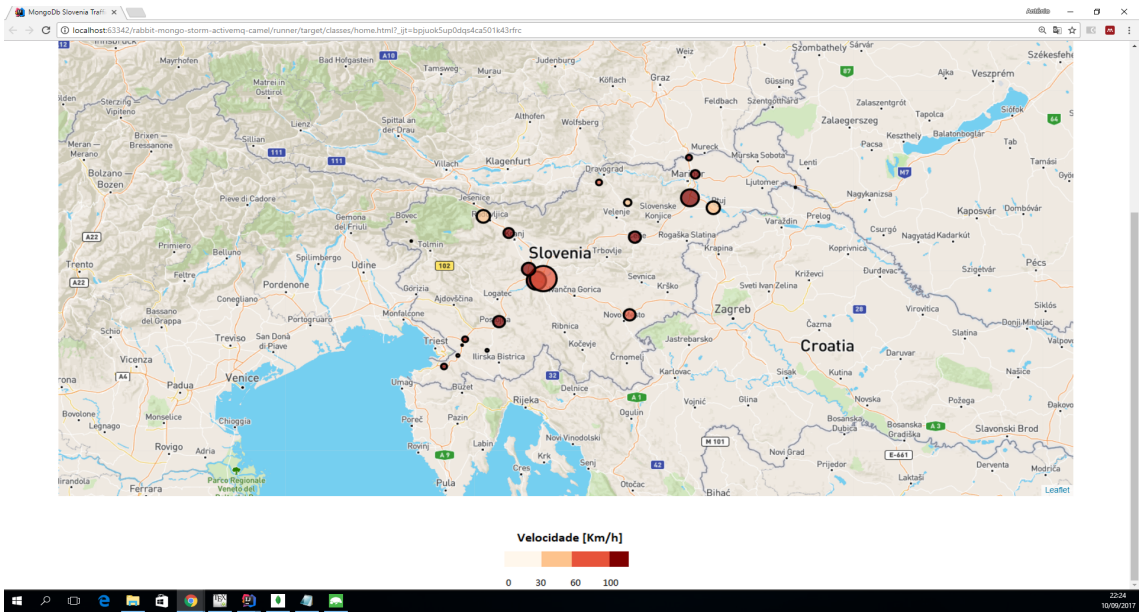


Figura 5.9: Visualização da ocupação e velocidade de um subconjunto de sensores na Eslovénia a partir da mudança de raio e cor de círculos, utilizando a API *Leaflet*

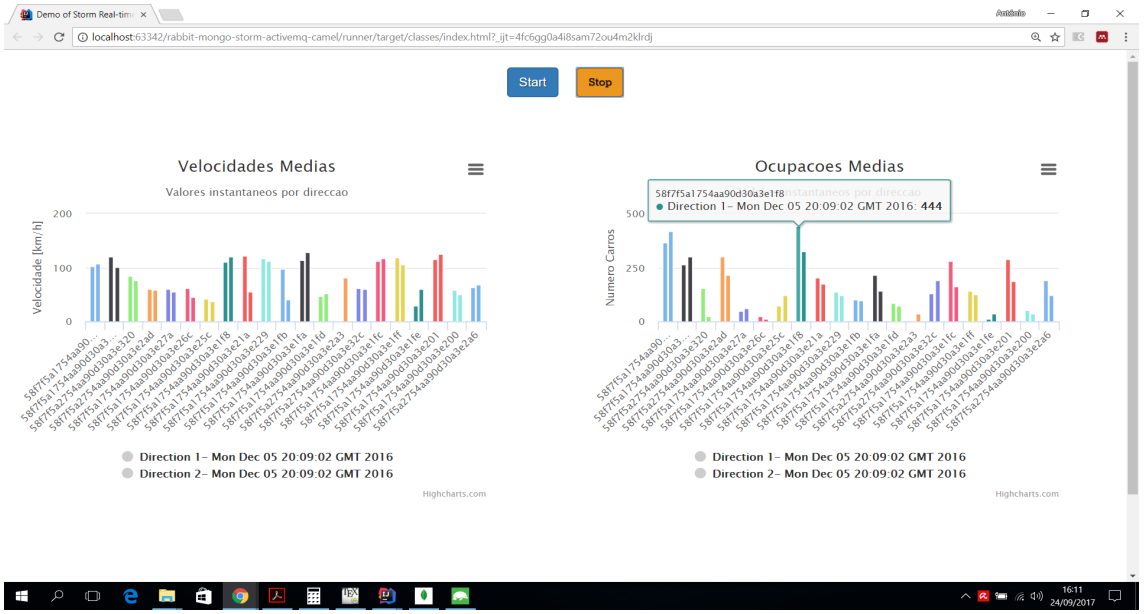


Figura 5.10: Visualização da comparação entre a ocupação e velocidade por direção de um subconjunto de sensores na Eslovénia , utilizando a API *Highcharts*

5.3 Discussão de Resultados

Comecemos pela análise dos resultados obtidos nos testes de eficiência ao RabbitMQ e ao Apache Storm.

No primeiro experimento com apenas 1 *worker* foram obtidos resultados de inserção de 8,8 mensagens por segundo, como ilustrado na Figura 5.2, e um consumo de 100% por parte dos 200 *spouts*/*consumers*, responsáveis pela ingestão de dados do Storm. Isto significa que temos uma entrada em média de 8,8 mensagens na topologia por segundo. No segundo experimento este facto repete-se, porém com um número ainda mais reduzido, 8,6 mensagens por segundo em média como ilustrado na Figura 5.4. Em ambos os casos, esta é a velocidade máxima de inserção tendo apenas uma *queue* em apenas um *node* do RabbitMQ, executado localmente. Estes valores são diferentes para cada vez que é feito o teste em *cluster*, pois para além da latência da escrita numa única fila no servidor do RabbitMQ, esta inserção também depende da latência do processo de *querying* à base de dados MongoDB e da serialização em objetos Java dos dados provenientes da mesma. Este número de mensagens inseridas, tão diminuto, faz com que uma ferramenta como o Storm, preparada para processar milhões de *tuples* por segundo por cada nó do *cluster* [4], seja desaproveitada da forma como a arquitetura da solução foi desenvolvida. Como podemos observar tanto na Figura 5.3 como na 5.5 as capacidades dos *bolts*, unidades responsáveis pelo processamento no Storm, tanto na configuração com 1 ou 4 *workers* registam valores muito perto de zero. O resultado deste parâmetro significa a percentagem de tempo que a unidade está ocupada a processar *tuples*, de 0 a 1, o que vem confirmar o desaproveitamento da eficiência do Storm, efetivamente a topologia pede um *rate* de entrada consideravelmente mais elevado.

Segundo os *guidelines* do Storm no que diz respeito ao *tuning* de performance da ferramenta, este índice é o principal para o qual se tem de olhar antes de se tomar decisões. Valores muito próximos ou superiores a 0,8 podem alterar o bom funcionamento da topologia e aumentar a latência no processamento de dados. Quando este facto acontece deve ser tomada a decisão de primeiro aumentar o número de *executors*/*threads* para o *bolt(s)* que apresentam o problema, e só em caso de prevalência do mesmo se deve aumentar o número de *workers*. No caso específico do primeiro teste realizado com apenas 1 *worker*, os valores de capacidade, como já referido, são aproximadamente zero em todas as instâncias de processamento, o que evidencia que não há necessidade de alterar nem o número de *threads* nem de *workers* para executar a topologia em questão. Podemos então concluir, que a ingestão de dados a partir do RabbitMQ da forma que está a ser realizada não tira partido da eficiência da ferramenta de processamento de *streams* utilizada, que depende diretamente desta ação.

Outro dos parâmetros que merece uma análise cuidada nas figuras que apresentam os resultados da execução em *cluster*, nas duas configurações, Figura 5.5 e 5.3 respetivamente, é o tempo de execução das unidades de processamento (*bolts*). Ao observarmos estes parâmetros podemos verificar os seus tempos de execução médios para processar um

tuple, e concluir que praticamente não se alteram em nenhum dos *bolts* da topologia com a modificação da configuração, ou seja, com o acrescento do número de *workers* não há alteração. Porém, apesar desta constatação, um resultado curioso é que o tempo total de processamento numa e noutra configuração difere, Figura 5.6.

A configuração com 1 *worker* em *cluster* obteve um tempo total de processamento de todas as mensagens em 3 horas 11 minutos e 30 segundos, enquanto que a configuração com 4 *workers* em 3 horas 34 minutos e 51 segundos, cerca de 23 minutos a mais. Estes foram os tempos obtidos para processar 95.449 mensagens de 17 meses de dados provenientes da base de dados MongoDB. Este resultado leva a uma interrogação bastante pertinente, pois se aumentamos o número de *workers* e consequentemente a capacidade de computação, como é possível que a configuração com 1 *worker* obtenha melhores resultados do que com 4 *workers*. Antes de mais nada, é preciso perceber deste tempo total obtido, qual a parte que corresponde ao tempo de inserção e qual corresponde ao tempo de processamento. No caso do primeiro teste com 1 *worker*, o *input rate* de mensagens é de 8,8 por segundo, o que significa que as 95.449 mensagens demoram cerca de 3 horas para serem consumidas, e o restante tempo de 11 minutos e 30 segundos pertence ao efetivo processamento das mesmas. No segundo teste, o *input rate* desce para 8,6 mensagens por segundo, o que dá um peso de 3 horas e 5 minutos à inserção e 29 minutos para o processamento do Storm. Quando comparamos o tempo de processamento do Apache Storm em ambos os testes percebemos que o teste em *cluster* com 4 *workers* impõe uma latência quase três vezes superior ao teste com 1 *worker*. Mas se o tempo de execução dos *bolts* é praticamente o mesmo nos dois casos, como pode haver esta discrepância?

A resposta é simples, quando definimos o número fixo de *workers* num *cluster*, estamos a fixar também um *worker process*/JVM com um limite máximo de memória alocada por *worker*. Esta memória limita o número de *executors*/*threads* que podem ser executadas em cada JVM, o que pode trazer problemas para a escalabilidade da topologia. O processo de aumento de *workers* serve para aumentar esta memória dividindo o trabalho por mais JVM. Este processo é realizado só e apenas quando a capacidade de alguma unidade de processamento se encontra a operar muito perto da sua capacidade máxima e esse não é o caso verificado nos ensaios realizados. Todavia, o aumento do número de *workers* pode ser considerado como uma ação de custo/benefício, pois apesar de libertar memória de cada JVM e consequentemente aumentar a velocidade de processamento acresce também alguma latência no que diz respeito à comunicação de *tuples* entre *workers*.

No caso de todos os *executors*/*threads* se encontrarem num só *worker*, os *tuples* viajam entre eles sem terem de atingir o *worker transfer buffer*, em vez disso, são depositados diretamente do *send buffer* para o *receiver buffer* do *executor* de destino. No caso em que temos o *executor* de destino na mesma máquina mas em outro *worker*, como é o caso do 2.º experimento, os *tuples* tem de seguir o caminho (*worker transfer buffer* → *local socket* → *worker receiver* → *executor receiver buffer*) o que aumenta a latência de processamento [4]. Esta latência é acrescentada ao tempo de processamento dos *bolts*, e por esta razão o tempo total de processamento do Storm é mais elevado quando utilizamos 4 *workers*

do que quando utilizamos 1, principalmente porque não há necessidade de libertar a memória das JVM, como é o caso. A latência desta transferência de *tuples* é ainda mais relevante pela versão do Storm utilizada, que tem muito piores resultados que a versão mais recente do framework, a versão 1.0.0. Esta conclusão proveio de um estudo [29], que compara as performances das duas versões, 0.9.1 com 1.0.0, e que chegou a resultados de 5 vezes menos latência no processo de troca de mensagens entre *workers* (*intra-worker*) para a versão mais recente do Apache Storm.

Em relação aos resultados obtidos para a visualização dos dados no browser podemos destacar quatro, ilustrados nas Figuras 5.7, 5.8, 5.9 e 5.10. Podemos perceber que este output da aplicação pode ser visto como uma simulação do que seria uma aplicação em tempo real para monitorização do tráfego na Eslovénia, atingindo um dos objetivos do desenvolvimento desta aplicação como já mencionado em capítulos anteriores. De facto, em tempo real podemos aferir informações sobre as zonas de maior congestionamento, horários de maior trânsito, ajudando por isso a perceber situações incomuns como acidentes, trabalhos na via que condicionam a velocidade e ocupação da estrada, entre muitos outros. É de realçar que a utilização de mapas, Figuras 5.8 e 5.9, dá uma visão mais espacial destas situações do que apenas utilizando gráficos, Figura 5.7 e 5.10.

Capítulo 6

Conclusão

Esta tese aborda a pesquisa no campo da mineração e processamento em tempo real de uma grande quantidade de dados. Vai desde a pesquisa de todas as técnicas/ferramentas e metodologias existentes até à implementação de um protótipo funcional que processa dados em tempo real. O cenário da aplicação situa-se no setor dos transportes inteligentes, um setor que tem vindo a mudar consideravelmente nos últimos anos e que necessita de plataformas capazes de processar a imensa quantidade de dados que está a ser produzida pelo mesmo em tempo real. Este tipo de processamento surge pela necessidade de prever, detetar situações e acompanhar o estado da rede rodoviária em tempo real, fornecendo assim aos utilizadores da via informações sobre incidentes tão graves como acidentes, estradas cortadas, congestionamento e até indisponibilidade por parte da rodovia, por exemplo devido a incêndios ou tempestades. Podemos então desta forma tomar decisões em tempo real que minimizem os impacto para a circulação de veículos na estrada.

O trabalho de pesquisa apresenta uma solução tecnológica juntamente com uma arquitetura de referência para uma escalável aplicação de processamento em tempo real de uma grande quantidade de dados. Apesar dos dados tratados serem de sensores localizados nas estradas da Eslovénia, este trabalho propõe um modelo base capaz de tratar dados de qualquer tipo de aplicação. A abordagem que foi apresentada contribui para uma melhor compreensão dos dados de trânsito disponíveis, bem como para a monitorização gráfica em tempo real do tráfego nas várias estradas do país em questão.

Um dos problemas que surgiu na implementação do processo de mineração de dados e que é necessário destacar foi a ocorrência de todo o *dataset* se encontrar em esloveno. Para além desta razão dificultar a compreensão dos dados, houveram algumas situações onde a falta de informação sobre os mesmos ou o desconhecimento dos hábitos culturais do país dificultou a sua análise. Um exemplo destas limitações é a não informação acerca de qual direção corresponde a numeração presente nos dados dos sensores (e.g.: 1 ou 2),

ou seja, qual a direção a que se refere cada número, pois analisar o tráfego na direção Liubliana-Maribor é diferente de o analisar no sentido contrário.

O principal desafio desta tese, no que diz respeito à implementação do protótipo foi a inserção de dados na ferramenta de processamento de *streams*. Efetivamente os dados disponíveis são estáticos, e sendo o objetivo da implementação simular uma aplicação em tempo real, os mesmos teriam de ser injetados de uma forma periódica na ferramenta, como se tivessem a ser produzidos no momento. A solução que se encontrou para colmatar este problema veio a relevar-se uma das principais condicionantes para a escalabilidade do protótipo. De facto, a ingestão de dados deveria acompanhar a capacidade de processamento do Storm e tal não acontece, sendo desaproveitado todo o seu poder de processamento, pois os dados que lhe chegam são insuficientes comparados com a sua capacidade de os processar. Não podemos assim considerar que o input de dados na ferramenta esteja efetivamente inserido no contexto de *Big Data*. A inserção é realizada apenas numa fila de receção de mensagens, executada em um só nó do servidor RabbitMQ, o que impõe latência ao processo de escrita e leitura das mesmas. Para além desta latência, a forma como os dados são retirados da base de dados e inseridos também atrasa o processo. Apesar disso, a execução do protótipo em tempo real representa uma aplicação muito válida para a visualização do estado do trânsito nas várias estradas na Eslovénia.

Outro dos grandes desafios deste trabalho foi a mineração dos dados. Na verdade, o grande volume e heterogeneidade dos mesmos dificultou todo o processo de análise, tendo sido realizadas várias operações de transformação. Por outro lado, a qualidade e disponibilidade dos dados podem afetar a veracidade de algumas conclusões retiradas na exploração realizada aos mesmos.

Em suma, os objetivos iniciais propostos foram cumpridos e esta tese contribui de várias formas para o tema estudado tais como: (i) implementação de uma metodologia de mineração de um grande volume de dados de trânsito; (ii) arquitetura para processamento e posterior visualização de dados em tempo real; (iii) implementação de um protótipo que valida a arquitetura referida.

6.1 Trabalho Futuro

Um dos principais melhoramentos à arquitetura é a forma como os dados estão a ser inseridos na ferramenta de processamento de *streams*. A estratégia de estar sempre a realizar consultas à base de dados, agregar os dados de todos os sensores da mesma por uma determinada data e hora antes de os enviar para a fila de mensagens consumida pelo Storm, talvez não tenha sido a melhor. Em vez disso, talvez fosse mais vantajoso colocar os dados tal como estão na base de dados na fila de mensagens e ter mais fases na topologia do Storm para os tratar e processar, computando as médias desejadas.

A visualização em tempo real dos dados após o processamento fez com que não fosse testada de todo a eficiência do Storm no processamento de dados em tempo real. Tal

acontece porque os dados têm de sair por ordem desse processamento, sob pena da visualização ilustrar dados desordenados, isto é, sem uma ordem específica de data e hora. Além disso, e como comum na maioria das aplicações de processamento em tempo real, o servidor do RabbitMQ teria de ser executado em *cluster*, com várias filas de mensagens a receber dados, para poder inserir dados a um ritmo competitivo para o Apache Storm. Outro dos melhoramentos seria, em vez da inserção de todos os dados numa só topologia, a divisão dos mesmos por várias topologias para obter uma melhor performance.

Por fim, outro dos possíveis aprimoramentos que podem também ser aplicados à arquitetura desenvolvida é a integração de restrições de segurança, como autenticação de dados, controlo de acesso e privacidade do cliente. Os problemas que podem estar associados a um sistema não seguro são enormes e para implementar este software num contexto industrial é fulcral que a segurança esteja garantida.

Este tipo de abordagens em relação ao tratamento de grandes volumes de dados serão cada vez mais comuns nos sistemas de transporte inteligentes. Até há bem pouco tempo, cada forma de transporte operava num sistema individual, isto é, os dados de cada um dos diferentes tipos de transporte não estava disponível para outros, e por isso não era possível obter uma visão global de todo o sistema de transportes. Atualmente, é possível os transportes operarem numa rede conectada e integrada, o que faz com que o volume, complexidade e heterogeneidade dos dados aumente e com que seja necessária a sua análise em tempo real. Desta forma, a utilização de ferramentas capazes de processar dados com uma latência muito reduzida será essencial para provocar uma real mudança na mobilidade.

Efetivamente este trabalho representa apenas uma peça do puzzle no que diz respeito à eficiência no processamento de dados em tempo real. Porém representa uma base sólida e que pode ser aprofundada no futuro em trabalhos que se foquem nesta temática.

Bibliografia

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C Erwin, E. Galvez, M Hatoun, A. Maskey, A. Rasin et al. "Aurora: a data stream management system". Em: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003.

[2] T. Abdessalem, R. Chiky, G. Hébrail e J. L. Vitti. "Using data stream management systems to analyze electric power consumption data". Em: *International Workshop on Data Stream Analysis (WDSA),. Caserta (Italie)*. Citeseer. 2007.

[3] *Apache Samza [Online]*. URL: http://samza.apache.org/.

[4] *Apache Storm [Online]*. URL: http://storm.apache.org/.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava e J. Widom. "Stream: The Stanford data stream management system". Em: *Data Stream Management*. Springer, 2016.

[6] J. Arbib e T. Seba. *Rethinking Transportation 2020–2030: The Disruption of Transportation and the Collapse of the Internal-Combustion Vehicle and Oil Industries*. 2017.

[7] H. Armelius e L. Hultkrantz. "The politico-economic link between public transport and road pricing: An ex-ante study of the Stockholm road-pricing trial". *Transport Policy* 13.2 (2006), pp. 162–172.

[8] B. Babcock, M. Datar, R. Motwani e L. O'Callaghan. *Sliding window computations over data streams*. Rel. téc. Stanford InfoLab, 2002.

[9] A. Bär, P. Casas, L. Golab e A. Finamore. "DBStream: an online aggregation, filtering and processing system for network traffic monitoring". Em: *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*. IEEE. 2014.

[10] I. I. for Business Value. "IBM Global Business Services" (2011). URL: http://www-935.ibm.com/services/multimedia/uk_en_transportation_and_economin_developement.pdf.

[11] M. W. Davidson, D. A. Haim e J. M. Radin. "Using networks to combine big data and traditional surveillance to improve influenza predictions". *Scientific reports* 5 (2015), p. 8154.

[12] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth e R. Uthurusamy. *Advances in knowledge discovery and data mining*. Vol. 21. AAAI press Menlo Park, 1996.

[13] P. Figueiras, R. Costa, G. Guerreiro, H. Antunes, A. Rosa e R. Jardim-Gonçalves. "User Interface Support for a Big ETL Data Processing Pipeline: An application scenario on highway toll charging models". *ICE Conference 2017* (2017).

[14] J. Gantz e D. Reinsel. "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east". *IDC iView: IDC Analyze the future* 2007.2012 (2012), pp. 1–16.

[15] L. Golab e M. T. Özsu. "Data stream management". *Synthesis Lectures on Data Management* 2.1 (2010).

[16] M. Hausenblas e N. Bijnens. "Lambda architecture". *URL: http://lambda-architecture.net/. Luettu* 6 (2015), p. 2014.

[17] G. Hebrail. "Data stream management and mining". *Mining massive data sets for security* (2008), pp. 89–102.

[18] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé et al. "IBM streams processing language: Analyzing Big Data in motion". *IBM Journal of Research and Development* 57.3/4 (2013), pp. 7–1.

[19] T. Hunter, T. Das, M. Zaharia, P. Abbeel e A. M. Bayen. "Large-scale estimation in cyberphysical systems using streaming data: a case study with arterial traffic estimation". *IEEE Transactions on Automation Science and Engineering* 10.4 (2013).

[20] IBM. "Delivering Intelligent Transport Systems". *Driving Integration and innovation* (2007). URL: http://www-935.ibm.com/services/us/igs/pdf/transport-systems-white-paper.pdf.

[21] S. Kamburugamuve, G. Fox, D. Leake e J. Qiu. "Survey of distributed stream processing for large stream sources". *Technical report* (2013).

[22] A. Kandel, M. Last e H. Bunke. *Data mining and computational intelligence*. Vol. 68. Physica, 2013.

[23] M. z. L. Golab. *Issues in Data Stream Management, Canada. SIGMOD Record, Vol. 32, No.2*. June 2003.

[24] *Land Transport Masterplan - Singapore*. 2013. URL: https://www.lta.gov.sg/content/dam/ltaweb/corp/PublicationsResearch/files/ReportNewsletter/LTMP2013Report.pdf.

[25] *MAPR*. URL: https://mapr.com/blog/stream-processing-everywhere-what-use/.

[26] L. Mearian. "By 2020, there will be 5,200 GB of data for every person on Earth" (2012). URL: http://www.computerworld.com/article/2493701/data-center/by-2020--there-will-be-5-200-gb-of-data-for-every-person-on-earth.html.

[27] E.-E. P. on Mobility Management. *Congestion charging in Europe*. 2015. URL: http://www.epomm.eu/newsletter/v2/content/2015/0415/doc/eupdate_en.pdf.

[28] "Monitoring Streams – A New Class of Data Management Applications" (2002).

[29] R. Naik. *Microbenchmarking Apache Storm 1.0 performance*. URL: https://br.hortonworks.com/blog/microbenchmarking-storm-1-0-performance/.

[30] *OPTIMUM*. URL: http://www.optimumproject.eu/.

[31] Z. Parker, S. Poe e S. V. Vrbsky. "Comparing NoSQL MongoDB to an SQL DB" (2013), p. 5.

[32] V. Pimentel e B. G. Nickerson. "Communicating and displaying real-time data with WebSocket". *IEEE Internet Computing* 16.4 (2012), pp. 45–53.

[33] K. N. Qureshi e A. H. Abdullah. "A survey on intelligent transportation systems". *Middle-East Journal of Scientific Research* 15.5 (2013), pp. 629–642.

[34] *RabbitMQ*. URL: https://www.rabbitmq.com/.

[35] A. Ramos, A. Rodrigues, A. Tsirimpa, A. Polydoropoulou, S. Machado, F. Antunes, P. Ventura e T. Garcia. "Proactive Charging Schemes for Freight Transport: Dynamic Toll Discounts as a Tool to Reduce the National Roads Traffic" ().

[36] A. Regalado. "The data made me do it". *Technology Review* (2013).

[37] *Republic of Slovenia - Statistical Office*. http://www.stat.si/StatWeb/en/News/Index/5227. Accessed: 2017-26-06.

[38] P. P. Sean T.Allen Matthew Jankowski. *Storm Applied*. Manning, 2014.

[39] K. Shvachko, H. Kuang, S. Radia e R. Chansler. "The Hadoop distributed file system". Em: *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE. 2010, pp. 1–10.

[40] *Spark Streaming [Online]*. URL: http://spark.apache.org/docs/latest/index.html.

[41] D. Sperling e D. Gordon. *Two billion cars: driving toward sustainability*. Oxford University Press, 2009.

[42] M. Stonebraker, U. Çetintemel e S. Zdonik. "The 8 requirements of real-time stream processing". *ACM SIGMOD Record* 34.4 (2005), pp. 42–47.

[43] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo e J. Eriksson. "VTrack: accurate, energy-aware road traffic delay estimation using mobile phones". Em: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2009, pp. 85–98.

[44] *TomTom Traffic Index*. https://www.tomtom.com/en_gb/trafficindex/city/ljubljana. Accessed: 2017-05-06.

[45] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham et al. "Storm@ twitter". Em: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156.

[46] R. Viereckl, D. Ahlemann e A. Koster. "Connected car report 2016: Opportunities, risk, and turmoil on the road to autonomous vehicles". *PwC, last accessed (16.01. 2017) at: http://www. strategyand. pwc. com/reports/connected-car-2016-study* (2016).

[47] R. Wirth e J. Hipp. "CRISP-DM: Towards a standard process model for data mining". Em: *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*. 2000, pp. 29–39.

[48] C. M. Wu, Y. F. Huang e J. Lee. "Comparisons between MongoDB and ms-SQL databases on the twc website". *American Journal of Software Engineering and Applications* 4.2 (2015), pp. 35–41.

[49] W. Xu, Z. Zhou, H. Zhou, W. Zhang e J. Xie. "MongoDB improves Big Data analysis performance on Electric Health Record system". Em: *International Conference on Life System Modeling and Simulation and International Conference on Intelligent Computing for Sustainable Energy and Environment*. Springer. 2014, pp. 350–357.

[50] J. S. Yi, Y. ah Kang e J. Stasko. "Toward a deeper understanding of the role of interaction in information visualization". *IEEE transactions on visualization and computer graphics* 13.6 (2007), pp. 1224–1231.

[51] J. Zhang, F.-Y. Wang, K. Wang, W.-H. Lin, X. Xu e C. Chen. "Data-driven intelligent transportation systems: A survey". *IEEE Transactions on Intelligent Transportation Systems* 12.4 (2011), pp. 1624–1639.

[52] X. Zhao, S. Garg, C. Queiroz e R. Buyya. "A Taxonomy and Survey of Stream Processing Systems" (2017).